AD-A026 452

# RANDOM VARIABLES AS A DATA TYPE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PREPARED FOR

ELECTRONIC SYSTEMS DIVISION

13 MAY 1976

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (*When Data Entered*)

| | | |
|---|---|---|
| **REPORT DOCUMENTATION PAGE** | | READ INSTRUCTIONS BEFORE COMPLETING FORM |
| 1. REPORT NUMBER ESD-TR-76-76 | 2. GOVT ACCESSION NO. | 3. RECIPIENT'S CATALOG NUMBER |
| 4. TITLE (*and Subtitle*) Random Variables as a Data Type | | 5. TYPE OF REPORT & PERIOD COVERED Technical Report |
| | | 6. PERFORMING ORG. REPORT NUMBER Technical Report 516 |
| 7. AUTHOR(*s*) Alan G. Nemeth | | 8. CONTRACT OR GRANT NUMBER(*s*) F19628-76-C-0002 |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS Lincoln Laboratory, M.I.T. P.O. Box 73 Lexington, MA 02173 | | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Program Element 62708E Project Code 6T10 |
| 11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, VA 22209 | | 12. REPORT DATE 13 May 1976 |
| | | 13. NUMBER OF PAGES 92 |
| 14. MONITORING AGENCY NAME & ADDRESS (*if different from Controlling Office*) Electronic Systems Division Hanscom AFB Bedford, MA 01731 | | 15. SECURITY CLASS. (*of this report*) Unclassified |
| | | 15a. DECLASSIFICATION DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT (*of this Report*)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (*of the abstract entered in Block 20, if different from Report*)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (*Continue on reverse side if necessary and identify by block number*)

random variables programming languages statistical compiler

20. ABSTRACT (*Continue on reverse side if necessary and identify by block number*)

This report discusses the use of random variables as a data type for programming languages. It demonstrates that for complex programs the results of the use of random variables are non-computable. After imposing restrictions on the class of programs to obtain a practical, although limited class of programs, we discuss the major problems of constructing a statistical compiler which accepts distributions for its input variables, and produces the distribution of its output variables. Both simplification rules and representation techniques for such a compiler are described. A simple example of such a compiler which has been implemented is described, and the problems in extending the implementation are explored.

Directions for future research work in this area and techniques for evaluating the utility of this approach are discussed.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (*When Data Entered*)

ESD-TR-76-76

194074

ADA026452

# Technical Report 516

A. G. Nemeth

# Random Variables as a Data Type

13 May 1976

Prepared for the Defense Advanced Research Projects Agency
under Electronic Systems Division Contract F19628-76-C-0002 by

**Lincoln Laboratory**

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LEXINGTON, MASSACHUSETTS

The work reported in this document was performed at Lincoln Laboratory, a center for research operated by Massachusetts Institute of Technology. This work was sponsored by the Defense Advanced Research Projects Agency under Air Force Contract F19628-76-C-0002 (ARPA Order 2929).

This technical report has been reviewed and is approved for publication.

FOR THE COMMANDER

Eugene C. Raabe

Eugene C. Raabe, Lt. Col., USAF
Chief, ESD Lincoln Laboratory Project Office

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LINCOLN LABORATORY

# RANDOM VARIABLES AS A DATA TYPE

*A. G. NEMETH*

*Group 24*

TECHNICAL REPORT 516

13 MAY 1976

LEXINGTON MASSACHUSETTS

# RANDOM VARIABLES AS A DATA TYPE*

## ABSTRACT

This report discusses the use of random variables as a data type for programming languages. It demonstrates that for complex programs the results of the use of random variables are non-computable. After imposing restrictions on the class of programs to obtain a practical, although limited class of programs, we discuss the major problems of constructing a statistical compiler which accepts distributions for its input variables, and produces the distribution of its output variables. Both simplification rules and representation techniques for such a compiler are described. A simple example of such a compiler which has been implemented is described, and the problems in extending the implementation are explored.

Directions for future research work in this area and techniques for evaluating the utility of this approach are discussed.

* This report is based on a thesis of the same title submitted to the Division of Engineering and Applied Physics at Harvard University on May 1976 in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

BLANK PAGE

CONTENTS

# RANDOM VARIABLES AS A DATA TYPE

## I. INTRODUCTION

Consider an airplane decelerating on a runway. Given the value of such variables as deceleration rate (perhaps time-dependent), touchdown speed, and touchdown point, one could write a program P to calculate the distance the plane had traveled down the runway before it reached a safe velocity for turnoff (perhaps 40 knots). Suppose that an airport designer has to place an exit taxiway at some distance down the runway. If the taxiway is too far down the runway, then planes will spend a longer time on the runway than they should, whereas, if it is not far enough, then many planes will not be at a safe velocity for turning by the time they reach the exit. The values of the touchdown speed and the location of the touchdown point are not the same for every plane which lands, but their distribution can be measured. Similarly, although the measurement problem is harder, one can measure the deceleration of planes on the runway or one can assume values from the known characteristics of various models of airplanes. The program P that was originally written describes how to calculate the distance at which to place the exit for some specific values of the input variables, but the information that the airport designer really wants is at what distance will say 95 percent of the planes be able to turn safely. (Presumably he will provide other exits for those unable to turn at this strategically placed exit.)

The techniques described in this report will allow the airport designer to come to the computer with the program which calculates the result for a particular set of values and information regarding the distributions of the set of values for the input variables. The result will be information regarding the distribution of the set of values for the output variables.

Thus the programmer can think of the behavior of one individual in the population of interest and write his programs to describe that behavior. Then, in a separate step, he can describe the relevant characteristics of his population using the appropriate statistical tools. The machine can then combine this information to obtain a description of the output population.

More specifically, this work aims toward an alternate mode of computation which would allow a user to request that his program be compiled into code which will accept statistical descriptions of his input variables and will produce, when executed, statistical descriptions of the output variables.

Formally then, given a program P (i.e., an ordered set of instructions for calculating a set of output values from a set of input values), I wish to "compile" that program to obtain another program P', whose inputs are statistical descriptions of the sets of values assumed by the inputs to P and whose output is a statistical description of the sets of values assumed by the outputs of P. We refer to the program P' as the statistical analog of P, and the process of producing this program will be referred to as a statistical compilation of P.

Another application of this work is in cost forecasting. Suppose that I am an entrepeneur thinking about selling widgets. The process used to produce widgets is not fully reliable, and in each batch of widgets produced only about 80 percent are acceptable. Moreover, this yield is variable; on some days, only 10 percent of the widgets will be acceptable, while on others 100 percent will be. Also, I know the current cost of raw materials and labor, but I want to be able to estimate what to charge for widgets a year from now when these costs will have changed. I am able to guess at distributions for these costs in a year (in the jargon of the decision analyst, I am willing to assess a prior judgmental distribution for these values). I would like to estimate what the manufacturing costs per widget produced will be. The mean value of these costs will allow me to calculate my expected profit, while the weight in the upper tail of the distribution will tell me my risk. Of course, two of my variables, namely yield and cost of materials and labor, are only known statistically. It is simple to write a program, assuming I know the yield and the costs of materials and labor, to obtain the manufacturing cost per widget. I would like to automatically combine this program with the distributions for yield and costs to obtain a distribution for the manufacturing cost per acceptable widget.

Another way to view this report is in terms of data types of variables in a programming language. Any number of standard data types exist in common programming languages (e.g., FORTRAN, PL/1). The extensible languages provide the programmer with mechanisms for providing additional data types as needed (see [Standish 75] for a survey of work in this area). What we are proposing in this thesis may be viewed as a new data type, which I shall call *random*. This may be used in conjunction with any of the standard types as in *real\random* or *int\random* or *real\array\random*. A variable of type *random* must be represented internally by some representation of the distribution for the variable. The internal representation might be in terms of the density function, the cumulative density function, or perhaps some alternate form such as moments. Moreover, the representation may be symbolic, i.e., the system implementor might choose to represent the random variable by a function which calculates the density function.

Regardless of which representation is chosen, the implementor may now proceed to define, in an extensible language environment, exactly what is meant by our new data types. For example, the sum of a *real* $x$ and a *real\random* $y$ would be defined to be of type *real\random*, and its representation would be the representation of a distribution which is that of $y$ but shifted by the amount $x$. Similarly, forming the sum of two *real\random* variables would cause the invocation of a procedure to calculate the representation for the convolution of their respective density functions.

However, as the implementor discovers fairly rapidly, the construction of such a system runs into many difficulties. Consider, for example, the following program (the syntax is that of EL1 [Wegbreit 74], an extensible language in use at Harvard)

```
DECL  X, Y, Z:  real\random;
DECL  A, B:  real\random;
A <- gaussian(0, 3);
     /* 'A is assigned a normal distribution of mean 0 and sigma of
3';
B <- gaussian(2, 4);
X <- A + B;
     /* 'X is thus a normal distribution of mean 2 and sigma of 5';
Y <- A + B;
Z <- X - Y;
     /* 'Z is properly a point distribution with probability 1 of having
the value 0';
```

Note that unless, at the time that $Z$ is being computed, the dependence of the distributions for $X$ and $Y$ is included in the computation, the result would be incorrect. Although $X$ and $Y$ are both *gaussian*(2,5), it is not correct to write $Z$ as *gaussian*(2,5) – *gaussian*(2,5) which would evaluate to *gaussian*$(0,5\sqrt{2})$. We assume that successive calls to *gaussian* produce distributions which are statistically independent. However, the variables $X$ and $Y$ are not statistically independent and this dependency causes the different result.

The thrust of this report is the extension and unification of a series of attacks by different authors on the problems of computing with data which are only known statistically. Because of the wide variety of statistical representations of data which are available, previous work has varied quite broadly in the approach to the problem, depending on the assumed representation. Thus, we have at one extreme, a description of a data value by the upper and lower bounds of the interval that contains it as in the interval arithmetic of Moore [Moore 66], while at the other extreme is Monte Carlo simulation where the entire distribution function of the data is used. Below we survey some of the more well-known attacks on this type of computation as well as some of the work which is closer in flavor to this report.

## Interval Analysis

The ideas expressed above may be viewed as a generalization of the interval arithmetic of Moore [Moore 66]. In his work, he limits the descriptions of his input variables to simply an upper and lower bound and attempts to calculate upper and lower bounds on the output variables which are as tight as possible. The automatic methods usually provided are a package of subroutines which are invoked to perform interval addition when addition is called for in the program, and similarly for interval subtraction, multiplication, and division. This results in a simple implementation of the interval scheme, but ignores information regarding the joint dependencies of variables. As a simple example of this consider the usual identity

$$(a + b)\,x = ax + bx$$

Suppose

$$a = [1 \quad 2]$$

$$b = [-3 \quad -2]$$

$$x = [3 \quad 5]$$

then

$$(a + b) = [-2 \quad 0]$$

$$(a + b)\,x = [-10 \quad 0]$$

whereas

$$ax = [3 \quad 10]$$

$$bx = [-15 \quad -6]$$

$$ax + bx = [-12 \quad 4]$$

Note that the reason that the right side provides such poor results is due to failure to take into account that x must have the same value both times it is evaluated. That is, ax and bx may not be evaluated independently (i.e., as if it were ax and by where x and y have the same interval description) if one is to obtain a tight bound on the result for ax + bx. In general, the failure to take cognizance of these dependencies broadens the bounds which are obtained by interval analysis techniques. One can show indeed that for any intervals I, J, K

$$I(J + K) \subset IJ + IK$$

although if $JK > 0$ (an interval is greater than zero if its lower bound is greater than zero), then equality holds. Depending on the extent of these dependencies, the bounds obtained may or may not be practically useful.

## Monte Carlo Simulation

Somewhat at the other extreme of what can be done are the techniques of Monte Carlo simulation [Hammersley 64]. Here the assumption is that distributions are provided for each of the input variables and sample sets of input values are then drawn from the input distributions. The resulting values are then used in the computation in order to obtain an output value. The process is repeated many times in order to build up the statistics of the output variable. By clever sampling, it is often possible to obtain the output distribution much more efficiently than "crude" sampling techniques would indicate. Still there are a number of limitations of the technique

(1) It requires a distribution for the input variables, rather than some reduced amount of information (i.e., the first four moments).

(2) It requires many executions of the program to build the desired measures of the output variables to the necessary precision.

(3) It requires generating random samples from what may be complex and interdependent distributions.

(4) The program is viewed as a black box, and any information regarding its structure is ignored.

Note however, that there are no problems due to dependencies between variables in the program. Since each execution of the program reflects the dependencies which exist, the resulting output values also accurately reflect those dependencies.

## Franson's Work

Franson, in a Master's thesis [Franson 69] at the Naval Postgraduate School in Monterey, California, formulates some basic rules for manipulating probability distributions to attack problems similar to the ones discussed here. The compilation techniques are essentially manual while the computation algorithms invoked are fairly straightforward numerical integrations of the convolution integrals obtained. In addition, the technique does not handle dependencies among the variables.

## LEADICS Simulator

A group at the University of Notre Dame has developed a simulation for the Law – Engineering Analysis of Delay in Court Systems (LEADICS) project [Sain 73]. They model each step through a criminal justice system by a representation of the probability distribution of the delays through that step. An individual leaving one step of the criminal justice system may directly proceed to the next step or pass through a multiway branch which has a probability associated with each path, or through a feedback path. Their simulator combines the distributions at each step to produce the delay distribution for the entire system. They represent the delay distribution by a rational fraction approximation to the characteristic function and manipulate it to represent the operations of addition, branching, and feedback.

Their simulation handles only a very restricted set of operations on the distributions and it is not obvious how to extend it. Moreover, they have completely ignored effects due to correlation between the distributions in their model; i.e., if the time from the commission of a crime to arrest is long, is the trial also likely to be long? In general, these possible correlations affect their results in an unknown fashion and must be included in a more careful analysis.

### Bērziņš Work

Of all the work I am aware of, a recent Master's thesis by Bērziņš [Bērziņš 75] represents the most sophisticated development in this area. His work was performed in the context of an effort to build an automatic programming system which will generate programs implementing business information systems (called Protosystem I). Part of this system is an optimizer for choosing data representations and program structures. This requires a "question answerer" which will supply information about the expected behavior of the information system.

His construction of such a question answerer by techniques which are cleverer than Monte Carlo analysis runs into many of the problems addressed in this thesis. His system consists of a number of pieces including an interval analysis package, a Monte Carlo simulator, and a package for the analysis of random variables by the manipulation of distributions. His work on random variables provides an implementation which differs from that described in this thesis in a number of ways: (1) In my terms, the class of base functions he has chosen are plus and a conditional of the form *if* $x > y$ *then* $u$ *else* $z$. However, when he needs the analysis of more complicated situations such as looping or multiplication, he uses the Monte Carlo simulator in the environment to obtain an approximate answer and then continues, and (2) his handling of dependent variables is by maintaining a measure of the degrees of freedom of his input variables (the "effective number") and using this at various points to improve his estimate of the answer. He clearly recognizes this as an inadequate solution to the problem for he states:

> [In an operation, if] the input variables are interdependent, then the correlations get very complicated and even though they are not necessarily small effects, I do not see a practical way of taking them into account. In these cases, I recommend treating the outputs as though they were independent even though they are not.
>
> It is probably a good idea to put in a test for the troublesome case, and to print a warning that an unsafe answer has been produced... [Bērziņš 75, pp. 199-200].

### Overview of Document

I begin by proving (Sec. II, Theorem 1) that the problem stated in its most general form is non-computable in the classic sense of being equivalent to the halting problem for Turing machines. One approach to circumventing this result is to reduce the class of programs which may be considered but provide exact techniques for that reduced class. It is this approach which is explored in this report. There are however other approaches possible. One such approach is to handle all functions with a weaker description of the random variables than their distribution as is done in interval analysis. Another approach is to accept some theoretical inaccuracies in the result as in Monte Carlo simulation.

Accordingly, we restrict our attention to a class of functions which may be described as trees (that is, they contain no loops at all). In Sec. II, I compare this class of functions to the various hierarchies of computational complexity of functions which have been explored in the literature.

In Sec. IV, I discuss the simplification rules for a statistical compiler, while Sec. V explores the variety of representation techniques which might be employed. These topics are the key design issues faced by the implementor of a statistical compiler.

In Sec. VI, I switch gears to a more experimental approach and describe an implementation of a prototype system for statistical computations which I have constructed in ECL at Harvard.

Section III states my approach to the problem of producing statistically analogous computations for any tree-type computation. Here I indicate why a compilation approach is necessary and delineate the key issues for constructing such a compiler. Section VII discusses what would be required to extend this work to provide a practical general-purpose system (i.e., a FORTRAN with random variables as a data type). Finally, Sec. VIII is some thoughts about future work in this area.

The main result is a new technique for attacking this type of problem. In addition, Theorem 1 in Sec. II appears here for the first time although the result is due to D. Tsitchritzis. The Rules 4 and 5 in Sec. IV are new here, while Rule 3, although discovered independently, is a restatement of already known results. The computational techniques in Sec. V which use moments are new to the computer science field, primarily because of the discussion of numerical errors in the solution of the moment problem, although these results have all appeared in the mathematical literature.

## II. PROBLEM REDUCTION

Let me begin with a more formal statement of the problem. I wish to construct a program C which accepts the following items as input:

- f – a program which is the computational description of some mathematical function with a real-valued result, drawn from a class F.
- $p_i$ – a set of programs which are computational descriptions of the probability distributions for the input variables of f (which I will assume independent for this section). I make the further assumption that the $p_i$ are discrete distributions, an assumption I shall relax shortly.
- z – a value chosen from the set of possible output values for f.

The value returned by $C(f, p_i, z)$ is the probability that the result of f will be less than or equal to z given that the probabilities of the input variables are as described by the $p_i$'s.

Note that this view of C is not intended to preclude possible partitions of the work performed by C. C might be implemented, for example, as a "statistical compiler" SC, whose input is merely f and whose output is an $f'(p_i, z)$ which when fed the $p_i$'s and z can complete the calculation. Thus, we might write $C(f, p_i, z) = [SC(f)]\ (p_i, z)$. Yet another partition of C provides a "Distribution Calculator" DC, whose input is f and the $p_i$'s and whose result is a computational description of the probability distribution for z. That is, $C(f, p_i, z) = [DC(f, p_i)]\ (z)$.

As a very simple example of what is intended here, consider $f(a, b) = a + b$. Recalling that the probability density function of the sum of two independent random variables is the convolution of the probability density functions of the random variables, we have SC producing a general convolver. That is, SC produces f' whose inputs are two functions and a specific value z and evaluates the convolution of those functions at the point z. DC, on the other hand, accepts as input the function f and in addition descriptions of $p_a$ and $p_b$ and produces the distribution of the result (perhaps using FFT techniques).

Assuming the class of functions F is sufficiently broad, we may prove that C cannot exist, regardless of which implementation form is chosen. Consider Kleene's predicate $T(x, y, z) = 0$ if Turing machine x starting with y on its input tape stops at exactly z steps; 1 otherwise. Consider a restricted form of Kleene's predicate $\lambda y. T(x, x, y) = f_x(y)$. That is, we look at a function which for a specific Turing machine x started with x on its input tape returns 0 if the machine will stop at exactly y steps and 1 otherwise. Then we have the following theorem:

**Theorem 1.** (D. Tsitchritzis)

If F includes, for every Turing machine x, $\lambda y. T(x, x, y)$ and $p_y$ is discrete, then C cannot produce correct output for all $f \in F$.

**Proof.**

Choose $p_y$ so that every positive integer n has some probability of occurrence. For example, such a choice might be $p_y(n) = 2^{-(n+1)}$ for positive integral n.

Now Turing machine x either halts with x on its input tape, or it doesn't. Suppose it doesn't. Then $C[\lambda y. T(x, x, y), p_y, 0]$ will be zero; that is, if the machine x never halts, then the probability of it halting at y steps for any y is zero. On the other hand, if Turing machine x halts after q steps, then $C[\lambda y. T(x, x, y), p_y, 0]$ will evaluate to $2^{-(q+1)}$. That is, if machine x halts at q steps, then the probability of $\lambda y.T(x, x, y)$ reporting its halting is $p_y(q)$, the probability assigned to the occurrence of the value q for the variable y. Therefore, the test comparing $C[\lambda y.T(x, x, y), p_y, 0]$ to 0 represents the test that Turing machine x ever halts. Thus, C represents a solution to the halting problem and therefore, cannot exist.

The extension to continuous distributions follows directly.

Corollary.

If F includes, for every Turing machine x, $\lambda y.T(x, x, \lfloor y)$ (where $\lfloor y$ is the "floor" of $y$, the greatest integer less than or equal to y), then C cannot produce correct output for all $f \epsilon F$.

Proof.

Same as the previous proof except choose $p_y$ to be a continuous distribution with positive weight in every interval $[i, i + 1)$ for every positive integer $i$.

As a matter of strategy, we must now consider what useful options are left to explore in spite of the negative result.

One difficulty was that we allowed a distribution which, although finitely describable, had positive weight at an infinite number of points. Were there only a finite number of points in $p_y$, the function C could be written as follows:

```
(A) Build a simulator for Turing machine x
(B) For q increasing by 1 do
    (C) Simulate step q
    (D) If Turing machine x halts, output $p_y(q)$
    (E) Accumulate $p_y(q)$ into p
    (F) If p equals 1, terminate; else, loop.
```

This must complete in a finite time, since only a finite number of values of q must be explored.

Similarly, if we are willing to accept a result within $\delta > 0$ of the true answer, where $\delta$ is given as an additional input to C, we can solve the problem in similar fashion. Merely change the test in step (F) above to p greater than $1 - \delta$. Since at some finite point, the weight of the tail of the $p_y$ distribution must be less than $\delta$, this process must also terminate.

Finally, another approach we may consider is to restrict the class F so as not to include $\lambda y.T(x, x, y)$ for all x. Since the proof depends on the use and properties of Kleene's predicate, the proof presented would not work in this case, and indeed, the remainder of the report will be devoted to describing techniques which may be employed for a class F which does not contain Kleene's predicate.

I choose this approach to attacking the problem for the following reasons:

1. Monte Carlo techniques very effectively attack the case where we are willing to get an answer which is within $\delta$ of the true answer, although the computation time increases without limit as $\delta$ approaches zero.

2. The restriction to a finite distribution allows some trivial techniques to be used which can be very expensive to compute. Any ways around these expensive computations that I have attempted to devise have not crucially depended upon the finiteness of the distribution. They are

thus liable to being applied to infinite distributions and are therefore, by Theorem 1, not applicable to complicated programs.

3. My restricted class of functions is still large enough to be interesting both theoretically and practically.

## Computation Trees

Therefore, we look for some restriction which will eliminate Kleene's predicate from our class of functions. We note that computing Kleene's predicate involves (1) simulation of a single step of any arbitrary Turing machine, (2) looping for the number of steps specified in the input, and (3) testing for a halting configuration of the Turing machine. Due to the simplicity of Turing machines, (1) and (3) are exceedingly simple to perform and any language which excludes them is probably uninteresting. In addition, during the preliminary work on this report, the language constructs which caused the largest amount of difficulty in developing algorithms to handle *random* variables were the constructs involved with looping, i.e., FOR, GO TO, WHILE, etc. Thus, we choose as our restricted language a computation form whose programs may be represented as trees (no loops or cycles are permitted) and call these computation trees.

Formally, given a base class of functions $\mathcal{F}$, we may talk about the class of computation trees $\Omega$ based on functions from $\mathcal{F}$. The nodes in the tree will indicate variables and the edges will indicate the dependence among variables. Each node which is not a terminal node will also indicate a function from the class $\mathcal{F}$ which is to be applied to the values assigned to the ancestors of this node to compute the value at the node. In addition, each node will indicate an ordering among its ancestor nodes to correspond to the arguments to the function. For simplicity, I will consider a single root node which is the output variable of the tree. Some typical trees are shown in Fig. II-1. We think of arcs in the tree as directed from the terminal nodes toward

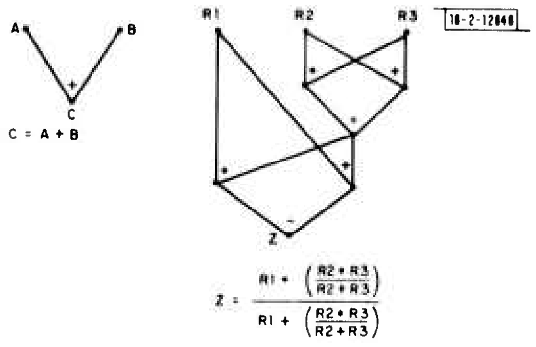
A
B
+
C
C = A + B
R1
R2
R3
18-2-12040
*
+
*
*
+
−
Z
Z = (R1 * ((R2 * R3)/(R2 + R3)))/(R1 + ((R2 * R3)/(R2 + R3)))

Fig. II-1.

the root. To each node, we may assign a level number which is the length of the longest path from any terminal node to that node. Then the computation to evaluate the output from a given set of input values proceeds as follows:

(1) Assign the input values to the corresponding terminal nodes in the tree and mark those nodes evaluated.

(2) Choose any unevaluated node whose level is less than or equal to the level of all other unevaluated nodes.

(3) Evaluate the chosen node by evaluating the function assigned to the node and using as arguments the values assigned to the ancestor nodes in the order indicated. Since the node chosen for evaluation is of lowest possible level, all nodes at lower levels have already been evaluated. Therefore, since a node's level must always be greater than the level of any of its ancestors all needed values are already evaluated.

(4) If the node just evaluated is the root node, then we are finished; otherwise, go back to step (2).

Since a tree contains only a finite number of nodes, the evaluation process must complete, assuming the evaluations of all the base functions terminate.

The class $\Omega$ then is the smallest class of functions which

(1) includes the functions of $\mathcal{F}$

(2) is closed under functional composition and selection.

The development in Sec. IV will be independent of the choice of the class $\mathcal{F}$ but will assume that we know how to perform certain basic operations related to the functions in $\mathcal{F}$. More specifically, if $f \epsilon \mathcal{F}$, we must know computational techniques for evaluating

$$p(\kappa) = \underset{\substack{a, b, c, \ldots \\ \ni f(a, b, c, \ldots) = k}}{\int \cdots \int} p(a, b, c, \ldots)\, da\, db\, dc \ldots \qquad (1)$$

where $p(a, b, c, \ldots)$ is either a product of input distributions or is the product of other similar computations. For example, if f in Eq. (1) is addition, then the operation intended is better known as convolution, while if f is subtraction then the operation is correlation. As f becomes more complicated, the computations invoked by Eq. (1) become quite difficult. We will refer to this type of computation throughout this report as f-convolution because of its similarity to the usual convolution operations.

In Sec. V, we will discuss how to compute (1) for several "simple" functions which we include in $\mathcal{F}$ (the constants, addition, subtraction, multiplication, etc.). As more computational techniques become available the class of functions which can be computed by the trees in $\Omega$ expands correspondingly. In particular, the inclusion of a conditional function of the form

$f(x,y,z) = \underline{\text{IF}}\ x > 0\ \underline{\text{THEN}}\ y\ \underline{\text{ELSE}}\ z$

would be particularly interesting since it would provide conditional expressions in $\Omega$. We suggest in Sec. V some computational devices for handling this.

Let us compare the possible classes $\Omega$ to some of the hierarchies of computational complexity which have been studied in the literature. This will help to elucidate the restrictions imposed on $\Omega$ by its definition and also will indicate which functions might be in $\mathcal{F}$. Meyer and Ritchie in a 1967 paper [Meyer 67] have studied the loop hierarchy $\{\mathcal{L}_n\}$. The class $\Omega$ cannot be extended to include all the elementary functions since Kleene's predicate is known to be elementary [Ritchie 63]. Thus, $\Omega \not\supset \mathcal{L}_2$ since $\mathcal{L}_2$ is the class of elementary functions. $\mathcal{L}_1$, however, may be characterized as the smallest class containing the constants, addition, proper subtraction, modulo, integer division, and closed under functional composition and conditional expressions.

If we can extend $\mathcal{F}$ to include the functions mentioned, then $\Omega \supset \mathcal{L}_1$. Moreover, it might well be possible to include exponentiation in $\mathcal{F}$ and this would result in $\Omega$ being greater than $\mathcal{L}_1$ since the exponentials are not in $\mathcal{L}_1$.

Another well-known hierarchy is the Grzegorczyk hierarchy $\{\mathcal{E}_n\}$. For $n > 2$, it has been shown [Meyer 67] to be equivalent to the loop hierarchy, i.e., $\mathcal{E}_n = \mathcal{L}_{n-1}$. In particular, then $\mathcal{E}_3$ is the elementary functions and thus $\Omega \not\supset \mathcal{E}_3$. $\mathcal{E}_2$ may be characterized [Cobham 64] as the smallest class containing the successor and multiplication functions, and closed under the operations of explicit transformation, composition, and limited recursion. While $\Omega$ includes all the base functions and the operation of explicit transformation and composition, it does not include the operation of limited recursion. Limited recursion is a much more potent combining rule than any allowed in the construction of $\Omega$. Still we are willing to allow more complex functions in our base class of functions $\mathcal{F}$ than are provided in the base set for $\mathcal{E}_2$. Moreover, since $\mathcal{E}_3$ may be produced by adding exponentiation to the class of base functions which generate $\mathcal{E}_2$, then exponentiation cannot be in $\mathcal{E}_2$. Since it might be possible to include exponentiation in $\mathcal{F}$, $\Omega$ may contain functions not in $\mathcal{E}_2$.

On the other hand, $\mathcal{E}_2$ contains the form of Kleene's predicate $\lambda y. T(x, x, y)$ which we have used [Ritchie 63]. Thus, $\Omega$ and $\mathcal{E}_2$ are not comparable. The basic difference may be summarized by the statement that $\Omega$ includes a more extensive set of base operations while $\mathcal{E}_2$ includes a more powerful combining rule.

## III. OVERVIEW OF THE TECHNICAL APPROACH

In Sec. I, I described a very broad class of problems and suggested that an object called a "statistical compiler" could be used in the solution of these problems. In Sec. II, I demonstrated that the class was too broad and described a restricted class of computations called computation trees which can be attacked with a statistical compiler approach. In this section, I want to indicate the major parts of a statistical compiler and describe the similarities and differences with conventional compilers.

There are a number of steps which must be performed to convert a program written in some language to code which can be executed by a computing machine. These steps indicated in Fig. III-1 involve the breaking down of the text of the program into basic units (lexical analysis), the combination of those units into well-formed expressions in the language (syntactic

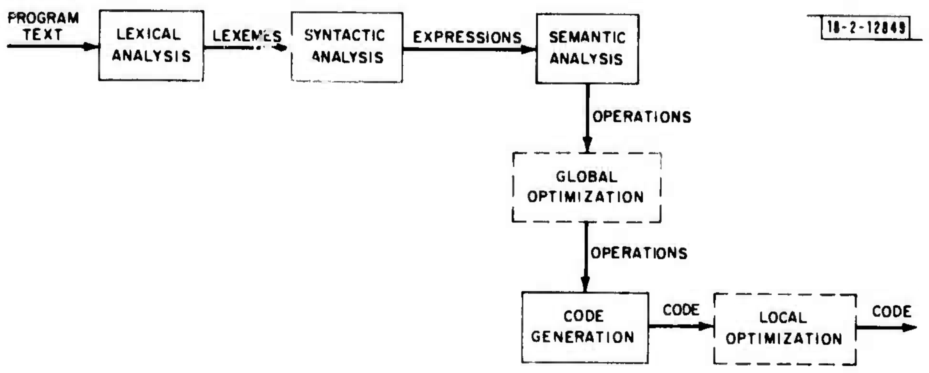

Fig. III-1. Major compiler phases.

analysis), extracting the intended operations from the expressions (semantic analysis), and producing the code to execute the program (code generation). In addition, one or two phases of code optimization may be employed: one between the semantic analysis and code generation, performing optimizations based on flow analysis (i.e., dead variable analysis, code motion) and another following or part of the code generation performing local machine-dependent optimizations (i.e., register allocation, elimination of unnecessary loads).

In any particular compiler, these phases may not be distinct in either time or code; however, all compilers must perform these basic operations to compile a program. An extensive discussion of code optimization techniques may be found in [Schaefer 73].

In a statistical compiler, the same major phases can be observed. Moreover, some of the phases are hardly modified at all, whether employed in a statistical or conventional compiler. The lexical analysis, syntactic analysis, and semantic analysis which can produce computation trees as described in Sec. II are straightforward applications of techniques currently in use [Aho 72].

On the other hand, the later phases of the compiler must be significantly different from a conventional compiler because the semantics associated with the operations in the computation trees are very different. Now, instead of addition as a basic operation, convolution becomes the basic operation. These changes clearly affect the code generation phase which must choose appropriate representations for the variables and produce code for the operations on those representations. The subject of efficient and accurate representation techniques is at the heart of constructing a statistical compiler and the issues are explored in Sec. V.

The need for optimization of the operations requested by the semantics is also apparent. Convolution operations are fairly expensive of computer time, thus any reductions that can be performed at compile time will pay off handsomely at run time.

I have indicated two distinct phases of optimization: global and local. Local optimizations which occur toward the tail end of code generation will deal with improving the code produced for specific cases, but are not prepared to deal with the important overall strategy questions which affect the amount of storage and execution time required in a first-order way.

As the small ECL example in Sec. I indicated, a key question in producing accurate results is the joint dependence of the variables entering into each step in the computation. But which joint distributions must we keep? Joint distribution functions will require large amounts of storage, and execution time, both probably increasing exponentially with the number of variables.

In the Sec. I example, only the joint distribution of X and Y is needed to accurately calculate the distribution for Z. However, without looking ahead to the use of X and Y, there is no way to know that one can afford to discard information about the joint distributions of X and A, or X and B. Indeed, if we did not know, at the time of execution of the statement $Y \leftarrow A + B$, the intended use for Y, we would be forced to calculate and maintain a representation of the joint distribution for the variables X, Y, A, and B.

Thus, we are led to the conclusion that any statistical compiler must employ optimization techniques aimed directly at reducing joint distributions by looking ahead at the use of the variables. Section IV describes a series of transformations to apply to the operations, referred to as the analogous program, which must be performed to "execute" a computation tree. These transformations start with an analogous program which maintains all possible joint dependencies, and demonstrates that certain reductions based on the analogous program and the computation tree can be performed without altering the distribution calculated for the output variable.

These simplification rules are then the equivalent, in a general way, to the usual global optimization techniques such as code motion, dead variable analysis, etc. The question now arises: when is a transformation a simplification? or, stated differently, which transformations should be applied to reduce the required effort at run time? The rules themselves merely guarantee that their application will not affect the accuracy of the result, but only the efficiency.

This question cannot be answered solely in terms of the transformations, since estimates of the cost of the analogous program are necessary. For example, it may well be necessary to know whether it is better to store a joint distribution of three variables or calculate ten convolutions. These questions obviously depend critically on the representation techniques which are employed.

The transformations described in Sec. IV are intended to be independent of the choice of representation, and thus no direct answer to these questions is presented in this report. Rather, the approach here is to describe the options for transformations in Sec. IV, the representation options in Sec. V, and indicate some specific ways of employing these in Secs. VI and VII.

The more general approach of attempting to describe the sequence of transformations to be performed based on cost functions for the analogous program has some basic flaws at this time. The first is that understanding the nature of these cost functions will require more experience with the representation techniques in Sec. V to reveal the basic properties common to all such cost functions. Further, the general statements which might be obtained by this approach, given the vague properties which can be ascribed to the cost functions in a representation independent fashion, lead me to the conclusion that these statements are likely to be vacuous. For example, although it is possible to suggest some hill-climbing approach to improving the analogous program, the available steps may well be so gross that no continuity is apparent.

For these reasons, I have chosen in this report to focus on the issues faced in the implementation of a particular statistical compiler, drawing from the general techniques in Secs. IV and V as necessary.

# IV. SIMPLIFICATION TECHNIQUES

In this section, we discuss rules which allow us to alter the analogous program for a particular tree in ways which preserve the correctness of the computation. Five such rules and their proofs are presented here. The reason for stopping at five is the simple one that I can think of no others which apply.

Throughout this chapter, we will make the noncritical simplifying assumption that our base functions are binary.

## Notation

We will often have occasion to refer to sets of nodes in a computation tree, and we will use the usual set notation $\{A, B, C, D\}$. Each node will be named by a letter of the capital alphabet and perhaps a subscript. Thus, a set of nodes might be denoted as $\{X_1, X_2, \ldots X_n\}$. We may abbreviate this as $\{\underline{X}\}$. The condition that node A has been evaluated to "a" during a particular evaluation of the computation tree will be denoted as $A = a$. Similarly, the condition for a set of nodes will be denoted $\{A = a,\ B = b,\ C = c,\ D = d\}$. It will often be convenient to abbreviate this, and we will use square brackets around the set to indicate it, thus $[A, B, C, D]$ is the same as $\{A = a,\ B = b,\ C = c,\ D = d\}$ and similarly $[\underline{X}] = \{X_1 = x_1,\ X_2 = x_2,\ X_3 = x_3, \ldots X_n = x_n\}$. The square brackets may be read as "the state of" and thought of in that manner. The brackets then identify an event in an event space which is specified by the contents of the brackets. Note the assumed correspondence between variables with upper-case names and variables with lower-case names.

4.1 Given a computation tree whose output variable is A, we will use $P\{A = a\}$ or $P[A]$ to denote the probability mass function which is the desired result when given the probability mass functions for the inputs. We will assume throughout that the terminal node distributions are statistically independent. Now, if the set of all nodes in the tree, excluding the root A is denoted by $\{X_1, X_2, \ldots X_k\}$ and $f_A$ is the function associated with A, and A's left and right ancestors are $X_i$ and $X_j$, respectively, then we may write the following trivial expression for $P\{A = a\}$:

$$(1) \qquad P[A] = \sum_{\substack{x_1, x_2, \ldots x_k \\ \ni f_A(x_i, x_j) = a}} P[X_1, X_2, \ldots X_k]$$

This expression is clearly not satisfactory from a computational viewpoint, although it is possible to utilize it to compute $P\{A = a\}$. The computation implied is to form the k-dimensional probability mass function by listing all the k-tuples for which the probability is nonzero (by enumerating all the possible combinations of the values of the input variables and evaluating all the intermediate results) and evaluating the probability mass function as the product of the probabilities of values of the independent input variables.

We may then always write trivially a program as in (1) for the computation of the probability mass function of the output. These programs are enormously expensive to compute as indicated above. Thus, we present a set of transformation rules which may be applied to such programs and which are guaranteed to maintain correct results for the computation. Applications of these rules may be used to transform a program into a broad variety of computational variants, some of which should be less costly to compute. In the next section, we will demonstrate the technique on a simple example program and in Sec. 4.3, we present formal statements of the rules and prove that their application does not change the value produced by the program.

## 4.2 Example

Consider the computation tree shown in Fig. IV-1 which represents the computation $Z = f_Z(f_A(f_X(W_1, W_2), f_Y(W_2, W_3)), f_B(f_Y(W_2, W_3), W_4))$. We may write quite generally that

$$(2) \qquad P[Z] = \sum_{\substack{w_1, w_2, w_3, w_4, \\ x, y, a, b \\ \ni f_Z(a, b) = z}} P[A, B, X, Y, W_1, W_2, W_3, W_4]$$

We may, by the definition of conditional probability, rewrite this as

$$(3) \qquad P[Z] = \sum_{\substack{w_1, w_2, w_3, w_4, \\ x, y, a, b \\ \ni f_Z(a, b) = z}} P[A, B, X, W_1, W_4 | W_2, W_3, Y]\, P[W_2, W_3, Y]$$

where we have chosen to condition on those nodes which are ancestors of both A and B. In this conditional distribution, A and B are now statistically independent, since we have fixed the values of their common ancestors, and their other terminal ancestors are independent. Thus, we may write,

$$(4) \qquad P[Z] = \sum_{\substack{w_1, w_2, w_3, w_4, \\ x, y, a, b \\ \ni f_Z(a, b) = z}} P[A, X, W_1 | W_2, W_3, Y]\, P[B, W_4 | W_2, W_3, Y] \times P[W_2, W_3, Y]$$

Consider the use of the random variable X in this equation. It appears only once and is summed over. Since

$$(5) \qquad \sum_x P[A, X, W_1 | W_2, W_3, Y] = P[A, W_1 | W_2, W_3, Y]$$

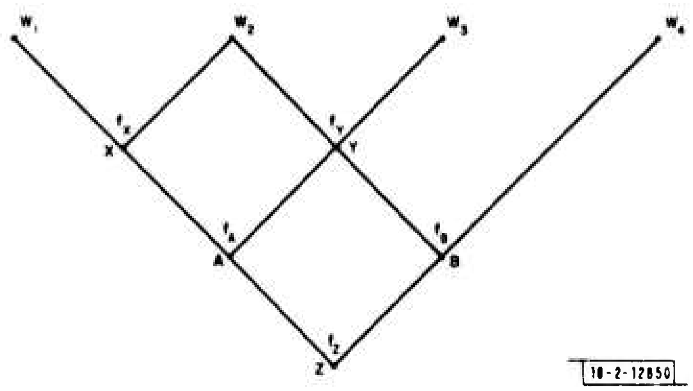

Fig. IV-1.

We may simplify (4) to obtain

(6) $$P[Z] = \sum_{\substack{w_1, w_2, w_3, w_4, \\ y, a, b \\ \ni f_Z(a,b) = z}} P[A, W_1 | W_2, W_3, Y]\, P[B, W_4 | W_2, W_3, Y] \times P[W_2, W_3, Y]$$

We may perform the same simplification on random variables $W_1$ and $W_4$ to obtain

(7) $$P[Z] = \sum_{\substack{w_2, w_3, \\ y, a, b \\ \ni f_Z(a,b) = z}} P[A | W_2, W_3, Y] \times P[B | W_2, W_3, Y]\, P[W_2, W_3, Y]$$

Note that in the original computation tree, if we break all the links which descend from Y, that $W_3$ no longer has a path to A. We say that $\{Y\}$ separates $\{A\}$ from $\{W_3\}$. Thus in the conditional distribution for A which is conditioned on $W_2$, $W_3$, and Y, we may eliminate $W_3$ from the condition because it provides no additional information to aid in evaluating the probability that A = a. Thus, we have

(8) $$P[Z] = \sum_{\substack{w_2, w_3, y, \\ a, b \\ \ni f_Z(a,b) = z}} P[A | W_2, Y] \times P[B | W_2, W_3, Y] \times P[W_2, W_3, Y]$$

Note that $\{Y\}$ separates $\{B\}$ from $\{W_2, W_3\}$ and thus we may write

(9) $$P[Z] = \sum_{\substack{w_2, w_3, y, \\ a, b \\ \ni f_Z(a,b) = z}} P[A | W_2, Y] \times P[B | Y] \times P[W_2, W_3, Y]$$

$W_3$ now appears only in the last factor and may thus be removed to obtain:

(10) $$P[Z] = \sum_{\substack{w_2, y, a, b \\ \ni f_Z(a,b) = z}} P[A | W_2, Y] \times P[B | Y] \times P[W_2, Y]$$

Now by using the definition of conditional probability we may rewrite this as

(11) $$P[Z] = \sum_{\substack{w_2, y, a, b \\ \ni f_Z(a,b) = z}} P[A, W_2, Y] \times P[B | Y]$$

Now $W_2$ appears only once and may be removed,

(12) $$P[Z] = \sum_{\substack{y, a, b \\ \ni f_Z(a,b) = z}} P[A, Y] \times P[B | Y]$$

Finally, by the definition of conditional probability, we have

(13) $$P[Z] = \sum_{\substack{y, a, b \\ \ni f_Z(a, b) = z}} P[A|Y] \times P[B|Y] \times P[Y]$$

Now the time to perform this calculation is proportional to the product of the number of nonzero probability values which are assumed by Y, A, and B, whereas the original expression (2) required time proportional to the product of the number of nonzero probability values for $W_1$, $W_2$, $W_3$, $W_4$. This assumes a simple-minded calculation scheme for the f-convolutions implied. If we assume that the number of nonzero probability values for $W_1$, $W_2$, $W_3$, and $W_4$ is the same number n, and further assume that the number of points in Y, A, and B are linear multiples of n, then the second computation involves $n^3$ operations as opposed to $n^4$. Whether these assumptions are met depends upon the spacing of the arguments in relation to the functions used. If points are spaced linearly and the functions are additive, then typically we will generate fewer than $n^2$ distinct points in forming the f-convolution. If all points are spaced evenly at the same intervals, the number of points in the resultant distribution is $2 \times n$.

4.3 We now state formally the rules which we have used to simplify the expressions in the example above.

1) Definition of Conditional Probability

(1) $$P[\underline{X}, \underline{Y}] = P[\underline{X}|\underline{Y}] \times P[\underline{Y}]$$

This is merely the usual statement of the definition of conditional probability. Note that if $P[\underline{Y}] = 0$ then the conditional probability is undefined.

2) Marginality of unique variables

If we have

(2) $$\sum_{\substack{\{\underline{z}\}, k \\ \ni B(\{\underline{z}\})}} P[K, \underline{X}|\underline{Y}] \times P[\underline{W}|\underline{I}] \times \ldots$$

Where B is a boolean condition not involving k, and k appears only once as shown, then we may rewrite as

(3) $$\sum_{\substack{\{\underline{z}\} \\ \ni B(\{\underline{z}\})}} P[\underline{X}|\underline{Y}] \times P[\underline{W}|\underline{I}] \times \ldots$$

where the $\{\underline{z}\}$'s represent all the small letter variables in $[\underline{X}]$, $[\underline{Y}]$, $[\underline{W}]$, $[\underline{I}]$, ..., and there may be as many more terms as desired.

The rule states that when a variable appears only once and is summed over, then we need not have the details of its joint distribution with the other variables in order to perform a correct calculation.

3) Conditional Independence

Definition: The least common ancestor set of A and B is the smallest set of nodes which

(1) are common ancestors, and

(2) has the property that any path from a common ancester of A or B passes through a node in the set.

If we have

(4) $P[A, B, \underline{Z}, \underline{X} | \underline{Y}]$

where $\{\underline{Y}\}$ includes all the least common ancestors of A and B, and $\{\underline{X}\}$ contains ancestors of A only, $\{\underline{Z}\}$ contains ancestors of B only, then we may rewrite this as

(5) $P[A, \underline{X} | \underline{Y}] \times P[B, \underline{Z} | \underline{Y}]$

This is true because the input variables are independent and once we are given the values for $\{\underline{Y}\}$, then there is no dependence of A and B on any common input variables and so they are independent.

<u>Proof of Conditional Independence</u>

Given a computation tree as shown in Fig. IV-2, divide all the inputs to the tree into four sets depending on whether they are ancestors of A, ancestors of B, ancestors of A and B, or ancestors of neither.

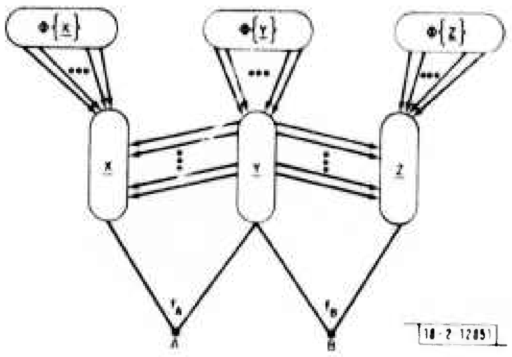

Fig. IV-2.

The set of input variables which contains ancestors of A only we will refer to as $\Phi\{\underline{X}\}$ since $\underline{X}$ includes all the ancestors of A only. Similarly $\Phi\{\underline{Y}\}$ is the set of input variables which consists of all the ancestors of A and B, and $\Phi\{\underline{Z}\}$ includes all the ancestors of B only.

The state of $\underline{Y}$, (i.e., the values of all the variables in $\underline{Y}$) is completely determined by the state of $\Phi\{\underline{Y}\}$. We will write this rather loosely as

(6) $[\underline{Y}] = f_Y([\Phi\{\underline{Y}\}])$

The state of $\{\underline{X}\}$ is determined not only by $[\Phi\{\underline{X}\}]$ but also by $[\underline{Y}]$ since $\{\underline{X}\}$ may have ancestors in $\{\underline{Y}\}$. Thus

(7) $[\underline{X}] = f_X([\Phi\{\underline{X}\}], [\underline{Y}])$

and similarly

(8) $[\underline{Z}] = f_Z([\underline{Y}], [\Phi\{\underline{Z}\}])$

Now we may write that

(9) $$P[A, \underline{X}, B, \underline{Y}, \underline{Z}] = \sum_{\substack{[\Phi\{\underline{Y}\}], [\Phi\{\underline{X}\}], \\ [\Phi\{\underline{Z}\}]}} P[\Phi\{\underline{Y}\}] \times P[\Phi\{\underline{X}\}] \times P[\Phi\{\underline{Z}\}]$$

where the sum is carried over all $[\Phi\{\underline{Y}\}]$, $[\Phi\{\underline{Z}\}]$ and $[\Phi\{\underline{X}\}]$ such that

$$\left.\begin{array}{l} f_A(x_i, y_j) = a \\ f_B(y_k, z_n) = b \\ [\underline{Y}] = f_Y([\Phi\{\underline{Y}\}]) \\ [\underline{X}] = f_X([\Phi\{\underline{X}\}], [\underline{Y}]) \\ [\underline{Z}] = f_Z([\underline{Y}], [\Phi\{\underline{Z}\}]) \end{array}\right\} \qquad \text{(I)}$$

Note that this statement is straightforward, since the complete states of $\{\underline{X}\}$, $\{\underline{Y}\}$, $\{\underline{Z}\}$, A, and B are determined fully by knowledge of the states of $\Phi\{\underline{X}\}$, $\Phi\{\underline{Y}\}$, and $\Phi\{\underline{Z}\}$. Further, for any state of $\Phi\{\underline{Y}\}$ meeting condition (I), the set of states of $\Phi\{\underline{X}\}$ and $\Phi\{\underline{Z}\}$ which also meet condition (I) is the same. This is because $[\underline{Y}]$ is fixed during the summation and the choice of valid $[\Phi\{\underline{X}\}]$ and $[\Phi\{\underline{Z}\}]$ is determined by $[\underline{Y}]$, $[\underline{X}]$, $[\underline{Z}]$ and not by $[\Phi\{\underline{Y}\}]$. Thus, since we have a complete cross product we may rewrite (9) as

(10) $$P[A, \underline{X}, B, \underline{Y}, \underline{Z}] = \sum_{[\Phi\{\underline{Y}\}]} P[\Phi\{\underline{Y}\}] \times \sum_{[\Phi\{\underline{X}\}], [\Phi\{\underline{Z}\}]} P[\Phi\{\underline{X}\}] \times P[\Phi\{\underline{Z}\}]$$

where

$$[\underline{Y}] = f_Y([\Phi\{\underline{Y}\}]) \qquad \text{(II)}$$

in the first sum and

$$\left.\begin{array}{l} f_A(x_i, y_j) = a \\ f_B(y_k, z_n) = b \\ [\underline{X}] = f_X([\Phi\{\underline{X}\}], [\underline{Y}]) \\ [\underline{Z}] = f_Z([\underline{Y}], [\Phi\{\underline{Z}\}]) \end{array}\right\} \qquad \text{(III)}$$

in the second sum.

Now, we may also write that

(11) $$P[\underline{Y}] = \sum_{[\Phi\{\underline{Y}\}]} P[\Phi\{\underline{Y}\}]$$

where the $[\Phi\{\underline{Y}\}]$ satisfy condition (II).

So we may then write that

(12) $$P[A, \underline{X}, B, \underline{Z} | \underline{Y}] = \sum_{[\Phi\{\underline{X}\}], [\Phi\{\underline{Z}\}]} P[\Phi\{\underline{X}\}] \times P[\Phi\{\underline{Z}\}]$$

where the sum is over all $[\Phi\{\underline{X}\}]$, $[\Phi\{\underline{Z}\}]$ which satisfy condition (III). If condition (II) is not met for any state of $\Phi\{\underline{Y}\}$ then the value for (12) is undefined.

Similarly, we obtain

(13) $$P[A, \underline{X} | \underline{Y}] = \sum_{[\Phi\{\underline{X}\}]} P[\Phi\{\underline{X}\}]$$

where

$$\left.\begin{array}{l} f_A(x_i, y_j) = a \\ [\underline{X}] = f_X([\Phi\{\underline{X}\}], [\underline{Y}]) \end{array}\right\} \qquad \text{(IV)}$$

or undefined if there is no $[\Phi\{\underline{Y}\}]$ such that $[\underline{Y}] = f_Y([\Phi\{\underline{Y}\}])$. Also,

$$(14) \qquad P[B, \underline{Z} | \underline{Y}] = \sum_{[\Phi\{\underline{Z}\}]} P[\Phi\{\underline{Z}\}]$$

where

$$\left.\begin{array}{l} f_B(y_k, z_n) = b \\ [\underline{Z}] = f_Z([\underline{Y}], [\Phi\{\underline{Z}\}]) \end{array}\right| \qquad (V)$$

and undefined if there is no $[\Phi\{\underline{Y}\}]$ such that $[\underline{Y}] = f_Y([\Phi\{\underline{Y}\}])$.

Now to complete the proof we need to show that

$$(15) \qquad P[A, \underline{X}, B, \underline{Z} | \underline{Y}] = P[A, \underline{X} | \underline{Y}] \times P[B, \underline{Z} | \underline{Y}]$$

When there is no $[\Phi\{\underline{Y}\}]$ such that $[\underline{Y}] = f_Y([\Phi\{\underline{Y}\}])$, then both sides are simultaneously undefined.

Otherwise, we wish to show that for any fixed $[\underline{Y}]$, $[\underline{X}]$, and $[\underline{Z}]$

$$(16) \qquad \sum_{[\Phi\{\underline{X}\}],\, [\Phi\{\underline{Z}\}]} P[\Phi\{\underline{X}\}] \times P[\Phi\{\underline{Z}\}] = \sum_{[\Phi\{\underline{X}\}]} P[\Phi\{\underline{X}\}] \times \sum_{[\Phi\{\underline{Z}\}]} P[\Phi\{\underline{Z}\}]$$

where in the first sum, condition (III) must be met, in the second sum condition (IV) must be met, and in the third sum condition (V) must be met.

Now, for any $[\Phi\{\underline{X}\}]$ which satisfies condition (IV), the set of states for $\Phi\{\underline{Z}\}$ which satisfies condition (III) is independent of the choice of $[\Phi\{\underline{X}\}]$. Again, since we have a complete cross product, the factoring of the sum is correct. Thus, (16) is an identity.

4) Separating Set Rule

If we have

$$(17) \qquad P[\underline{X} | \underline{Y}, \underline{W}] \times P[\underline{Z}, \underline{Y}, \underline{W}]$$

where $\{\underline{Y}\}$ separates $\{\underline{X}\}$ from $\{\underline{W}\}$ (i.e., any path from $W_i$ to $X_j$ includes at least one $Y_k$) and $\{\underline{X}\}$ has no ancestors in common with $\{\underline{W}\}$ not already included in $\{\underline{W}\}$, and $\{\underline{Z}\}$ is disjoint from $\{\underline{Y}\}$ and $\{\underline{W}\}$, then we may rewrite as

$$(18) \qquad P[\underline{X} | \underline{Y}] \times P[\underline{Z}, \underline{Y}, \underline{W}]$$

This is true since $[\underline{Y}]$ contains all the information in $[\underline{W}]$ which is used in calculating $[\underline{X}]$, so that the additional information embodied in $[\underline{W}]$ cannot change the probability of $[\underline{X}]$.

<u>Proof of separating set rule</u> (see Fig. IV-3)

As in the proof of conditional independence, we have sets $\Phi\{\underline{X}\}$, $\Phi\{\underline{Y}\}$, and $\Phi\{\underline{W}\}$ which are the set of input variables which feed the sets $\{\underline{X}\}$, $\{\underline{Y}\}$, and $\{\underline{W}\}$, respectively. To be more precise, $\Phi\{\underline{W}\}$ includes all those input variables which are ancestors of elements in $\{\underline{W}\}$. $\Phi\{\underline{Y}\}$ includes all those input variables which are ancestors of elements in $\{\underline{Y}\}$ and not included in $\Phi\{\underline{W}\}$. $\Phi\{\underline{X}\}$ similarly includes all those input variables which are ancestors of elements in $\{\underline{X}\}$ and are not included in either $\Phi\{\underline{W}\}$ or $\Phi\{\underline{Y}\}$. Then we may write

$$[\underline{W}] = f_W([\Phi\{\underline{W}\}])$$
$$[\underline{Y}] = f_Y([\underline{W}], [\Phi\{\underline{Y}\}])$$
$$[\underline{X}] = f_X([\underline{Y}], [\Phi\{\underline{X}\}])$$

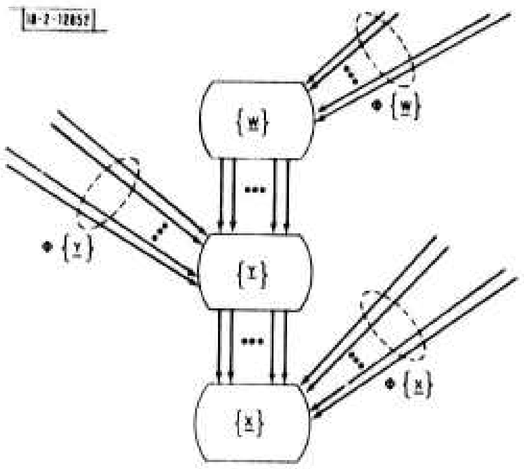

Fig. IV-3.

By assumption, the sets $\Phi\{\underline{X}\}$, $\Phi\{\underline{Y}\}$, and $\Phi\{\underline{W}\}$ are disjoint sets of statistieally independent variables and so we may write

(19) $$P[\underline{X}, \underline{Y}, \underline{W}] = \sum_{\substack{[\Phi\{\underline{X}\}], [\Phi\{\underline{Y}\}], \\ [\Phi\{\underline{W}\}]}} P[\Phi\{\underline{X}\}] \times P[\Phi\{\underline{Y}\}] \times P[\Phi\{\underline{W}\}]$$

where the sum is over all $[\Phi\{\underline{X}\}]$, $[\Phi\{\underline{Y}\}]$ and $[\Phi\{\underline{W}\}]$ such that

$$\left.\begin{aligned} [\underline{W}] &= f_W([\Phi\{\underline{W}\}]) \\ [\underline{Y}] &= f_Y([\underline{W}], [\Phi\{\underline{Y}\}]) \\ [\underline{X}] &= f_X([\underline{Y}], [\Phi\{\underline{X}\}]) \end{aligned}\right\} \qquad \text{(VI)}$$

This sum may be written as

(20) $$P[\underline{X}, \underline{Y}, \underline{W}] = \sum_{[\Phi\{\underline{X}\}]} P[\Phi\{\underline{X}\}] \times \sum_{[\Phi\{\underline{Y}\}]} P[\Phi\{\underline{Y}\}] \times \sum_{[\Phi\{\underline{W}\}]} P[\Phi\{\underline{W}\}]$$

sinee for any $[\Phi\{\underline{X}\}]$ meeting eondition (VI), the sets of states of $\Phi\{\underline{Y}\}$ and $\Phi\{\underline{W}\}$ meeting eondition (VI) do not ehange and similarly for $\Phi\{\underline{Y}\}$ and $\Phi\{\underline{W}\}$. Similarly,

(21) $$P[\underline{Y}, \underline{W}] = \sum_{[\Phi\{\underline{Y}\}]} P[\Phi\{\underline{Y}\}] \times \sum_{[\Phi\{\underline{W}\}]} P[\Phi\{\underline{W}\}]$$

where

$$\left.\begin{aligned} [\underline{W}] &= f_W([\Phi\{\underline{W}\}]) \\ [\underline{Y}] &= f_Y([\underline{W}], [\Phi\{\underline{Y}\}]) \end{aligned}\right\} \qquad \text{(VII)}$$

Thus,

(22) $$P[\underline{X} | \underline{Y}, \underline{W}] = \sum_{[\Phi\{\underline{X}\}]} P[\Phi\{\underline{X}\}]$$

where for some $[\Phi\{\underline{Y}\}]$ and $[\Phi\{\underline{W}\}]$, condition (VII) is met and $[\underline{X}] = f_X([\underline{Y}], [\Phi\{\underline{X}\}])$ and undefined if condition (VII) is not met for some $[\Phi\{\underline{Y}\}]$ and $[\Phi\{\underline{W}\}]$. Now

(23) $$P[\underline{X}, \underline{Y}] = \sum_{[\Phi\{\underline{X}\}]} P[\Phi\{\underline{X}\}] \times \sum_{[\Phi\{\underline{Y}\}], [\underline{W}]} P[\Phi\{\underline{Y}\}] \times P[\underline{W}]$$

where

$$\left.\begin{array}{l} [\underline{Y}] = f_Y([\underline{W}], [\Phi\{\underline{Y}\}]) \\ [\underline{X}] = f_X([\underline{Y}], [\Phi\{\underline{X}\}]) \end{array}\right| \qquad \text{(VIII)}$$

and

(24) $$P[\underline{Y}] = \sum_{[\Phi\{\underline{Y}\}], [\underline{W}]} P[\Phi\{\underline{Y}\}] \times P[\underline{W}]$$

where

$$[\underline{Y}] = f_Y([\underline{W}], [\Phi\{\underline{Y}\}]) \qquad \text{(IX)}$$

So

(25) $$P[\underline{X} | \underline{Y}] = \sum_{[\Phi\{\underline{X}\}]} P[\Phi\{\underline{X}\}]$$

where

$$[\underline{X}] = f_X([\underline{Y}], [\Phi\{\underline{X}\}]) \qquad \text{(X)}$$

and undefined if there is no $[\Phi\{\underline{Y}\}]$ such that (IX) is met.

This is identical with (22) except in the case that $P[\underline{Y}, \underline{W}] = 0$ and this case is irrelevant since both sides are multiplied by $P[\underline{Z}, \underline{Y}, \underline{W}]$.

5) Elimination of Complicating Variables

If we have

(26) $$P[\underline{X} | \underline{Y}, \underline{W}] \times P[\underline{Y}, \underline{W}, \underline{Z}]$$

where all the ancestors of $\{\underline{Y}\}$ are contained in $\{\underline{W}\}$, and $\{\underline{Z}\}$ is disjoint from $\{\underline{Y}\}$ and $\{\underline{W}\}$, and $\{\underline{X}\}$, $\{\underline{Y}\}$ and $\{\underline{W}\}$ are disjoint, then we may rewrite as follows:

(27) $$P[\underline{X} | \underline{W}] \times P[\underline{Y}, \underline{W}, \underline{Z}]$$

Proof (see Fig. IV-4)

Let $\Phi\{\underline{W}\}$ be the set of all input variables which are ancestors of nodes in $\{\underline{W}\}$, and let $\Phi\{\underline{X}\}$ be the set of input variables which are ancestors of nodes in $\{\underline{X}\}$ and not already included in $\Phi\{\underline{W}\}$. Then we have

$$\begin{aligned} [\underline{W}] &= f_W([\Phi\{\underline{W}\}]) \\ [\underline{Y}] &= f_Y([\underline{W}]) \\ [\underline{X}] &= f_X([\underline{W}], [\underline{Y}], [\Phi\{\underline{X}\}]) \end{aligned}$$

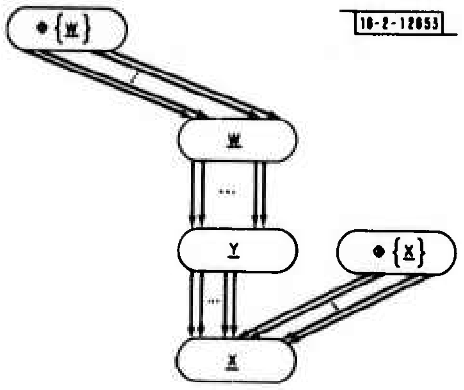

Fig. IV-4.

So we have trivially that

$$(28) \qquad P[\underline{X}, \underline{Y}, \underline{W}] = \sum_{[\Phi\{\underline{W}\}],\, [\Phi\{\underline{X}\}]} P[\Phi\{\underline{W}\}] \times P[\Phi\{\underline{X}\}]$$

where

$$\left.\begin{aligned} [\underline{W}] &= f_W([\Phi\{\underline{W}\}]) \\ [\underline{X}] &= f_X([\underline{W}], [\underline{Y}], [\Phi\{\underline{X}\}]) \end{aligned}\right\} \qquad \text{(XI)}$$

We may also write

$$(29) \qquad P[\underline{Y}, \underline{W}] = \sum_{[\Phi\{\underline{W}\}]} P[\Phi\{\underline{W}\}]$$

where

$$\left.\begin{aligned} [\underline{W}] &= f_W([\Phi\{\underline{W}\}]) \\ [\underline{Y}] &= f_Y([\underline{W}]) \end{aligned}\right\} \qquad \text{(XII)}$$

So we have that

$$(30) \qquad P[\underline{X} \mid \underline{Y}, \underline{W}] = \frac{\displaystyle\sum_{[\Phi\{\underline{W}\}],\, [\Phi\{\underline{X}\}]} P[\Phi\{\underline{W}\}] \times P[\Phi\{\underline{X}\}]}{\displaystyle\sum_{[\Phi\{\underline{W}\}]} P[\Phi\{\underline{W}\}]}$$

or undefined if there is no $\Phi\{\underline{W}\}$ such that (XII) is met. Now, we may similarly write that

$$(31) \qquad P[\underline{X}, \underline{W}] = \sum_{[\Phi\{\underline{W}\}],\, [\Phi\{\underline{X}\}]} P[\Phi\{\underline{W}\}] \times P[\Phi\{\underline{X}\}]$$

where

$$\left.\begin{aligned} [\underline{W}] &= f_W([\Phi\{\underline{W}\}]) \\ [\underline{X}] &= f_X([\underline{W}], f_Y([\underline{W}]), [\Phi\{\underline{X}\}]) \end{aligned}\right\} \qquad \text{(XIII)}$$

and

$$(32) \qquad P[\underline{W}] = \sum_{[\Phi\{\underline{W}\}]} P[\Phi\{\underline{W}\}]$$

where

$$[\underline{W}] = f_W([\Phi\{\underline{W}\}]) \qquad \text{(XIV)}$$

Thus, we obtain

$$\text{(32)} \qquad P[\underline{X} | \underline{W}] = \frac{\displaystyle\sum_{[\Phi\{\underline{W}\}], [\Phi\{\underline{X}\}]} P[\Phi\{\underline{W}\}] \times P[\Phi\{\underline{X}\}]}{\displaystyle\sum_{[\Phi\{\underline{W}\}]} P[\Phi\{\underline{W}\}]}$$

or undefined if there is no $\Phi\{\underline{W}\}$ satisfying condition (XIV). Comparing this to (30), we find the expressions are identical, except when $P[\underline{Y}, \underline{W}]$ is zero. But in that case the values are irrelevant since both will be multiplied by zero.

4.4 We now indicate that the transformations indicated in the example are all instances of applications of the rules just presented. Referring back to Sec. IV-2, we find that we move from (2) to (3) by the application of Rule 1 with $\underline{X} = \{A, B, X, W_1, W_4\}$ and $\underline{Y} = \{W_2, W_3, Y\}$. We move from (3) to (4) by Rule 3 with $\underline{X} = \{X, W_1\}$, $\underline{Z} = \{W_4\}$, $\underline{Y} = \{W_2, W_3, Y\}$. We move from (4) to (7) by three applications of Rule 2 to the variables X, $W_1$, and $W_4$. This can be pursued similarly throughout the example.

## Conclusion

The transformation rules described above must be applied in some sequence in order to simplify the statistical compilation. The sequence of application is in turn determined by the cost of various possible statistical compilation.

In the next section, we will discuss the various representation choices which may be employed by the implementor of a statistical computer. The implementor will have to decide which of these transformations should be used to improve the running time of the programs based on the different representations employed.

## V. REPRESENTATION TECHNIQUES

A key issue which will face the implementor of any statistical compiler is how to represent the random variables in his implementation. This section is a survey of possible answers to that question and is not intended to be complete; rather, it tries to give an indication of the range of possible choices and some of the effects of these choices on the efficiency and accuracy of the computations.

In choosing a representation for random variables, two key design parameters must be explored. The first of these is representational freedom, i.e., what range of distributions functions can be described accurately? For example, if a user arrives with an empirically determined distribution, how accurately can this distribution be described to the system? As the representational freedom is narrowed, fewer parameters are needed to select a particular distribution and storage space is economized. Moreover, certain calculations on the random variables represented by these distributions also become simpler. For example, if the chosen representation is by gaussian distributions, then addition of the random variables is simple to perform, but very few distributions can be accurately portrayed.

Another design parameter is the class of base functions (referred to in Sec. II as $\mathcal{F}$) which determines the class of computation trees $\Omega$ that the user may construct. As we indicated in Sec. II, in order to include a function f in $\mathcal{F}$, we must know computational techniques for evaluating the representation of f(X, Y) where X and Y are random variables with representations from the chosen representation class. Moreover, the techniques must extend to handle the cases where X and Y are not independent but rather a joint distribution is represented. We refer to this class of computations throughout this report as f-convolution because of its similarity to the usual convolution operation. Indeed, when the representation of distributions is in terms of their probability density functions, '+' – convolution of independent random variables is precisely the standard convolution operation while '–' – convolution is better known as correlation.

The actual arithmetic operations which are performed in the computer to calculate an f-convolution are strongly affected by the chosen representation and may be either simple or difficult for a particular f depending on the representation. Moreover, given a particular class of operations, the class may not be closed under our f-convolution operation. Thus if $R_X$ and $R_Y$ are the representations of the random variables X and Y, the calculation of the representation for f(X, Y) involves two steps: (1) calculation of an exact result for f(X*, Y*), where X* is the variable exactly represented by $R_X$, and (2) selection of a representative from the representation class for $R_{f(X, Y)}$. Both steps represent potential sources of errors which must be included in an analysis of the accuracy of the results of choosing a representation, although for some representations, which are closed under f-convolution, the second step is not an issue.

For example, if we choose addition as a member of the class of base functions, then it is unreasonable to choose a representation which is limited to uniform distributions. The representation of the sum of two independent random variables, each of which is represented by a uniform distribution, is never a uniform (except if one distribution is the degenerate case of a point distribution), and a uniform approximation would entail a significant loss of accuracy. See Fig. V-1 for an example of this.

Statisticians have, of course, spent much time and effort in developing techniques to describe distributions. They have not, as a rule, concerned themselves with representations

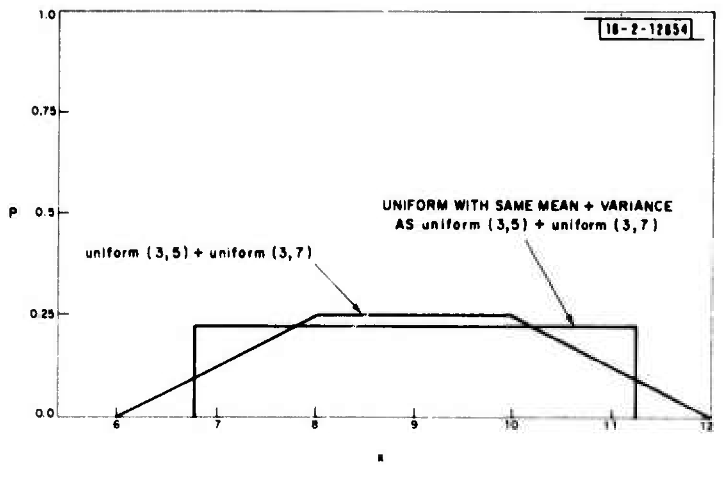

Fig. V-1.

which are convenient for f-convolution operations as necessary here. It is our intent to briefly indicate some of the statistical approaches to the representation problem and the characteristics of these representations for statistical compilers.

We will assume, except where explicitly stated otherwise, that the computer representation of REALs is exact. While this is, of course, not true, it is a much smaller source of error in an implementation than the other errors discussed in this section. Moreover, extended precision arithmetic can be used to correct this error in well-known ways, whereas the other error sources are not as easily controlled.

## Random Variables

We begin by reviewing what is meant by a random variable. A typical definition may be found in Feller [Feller 57]. We must first have a probability space which is constructed of elements. These elements are the basis for an algebra of sets (a $\sigma$-algebra) and a probability measure $P$ on those sets. Then we have

> Definition: A random variable is a function $\mu$ on a probability space such that for each real $t$ the set of points $x$ where $\mu(x) \leq t$ belongs to the underlying $\sigma$-algebra.

The definition basically insures that any real function which provides a well-defined distribution function is a random variable. That is, the function $F(t) = P(\mu(x) \leq t)$ is guaranteed to be well-defined by the definition of random variable.

Despite the generality of the definition, a number of properties of distribution functions can be directly determined from the properties of the $\sigma$-algebra. For example, the Jordan and Lebesgue decomposition theorems state that any distribution function $F$ can be expressed as a mixture of three types of distribution functions.

$$F = pF_{AC} + qF_S + rF_A$$

where $p \geq 0$, $q \geq 0$, $r \geq 0$, and $p + q + r = 1$. $F_{AC}$ is absolutely continuous, $F_S$ is continuous but singular (i.e., concentrated on a set of measure 0), and $F_A$ is atomic (i.e., concentrated on the set of its atoms, where atoms are single points with a positive probability weight).

For our work, we will assume that  = 0; that is, our distribution functions do not include components which are continuous on singular sets. We regard these as curiosities not likely to be observed in general practice.

We will look separately at absolutely continuous distribution functions and atomic distribution functions and the representations appropriate to each type. We can then explicitly represent any more general distributions as a mixture of the two.

## Absolutely Continuous Distributions

We will start with the absolutely continuous case. If F is a distribution function which is absolutely continuous, then there is an associated probability density function (pdf) f which exists almost everywhere such that

$$F(x) = \int_{-\infty}^{x} f(u)\,du$$

NOTE: We use upper case letters to denote distribution functions, and lower case letters to represent density functions.

Thus, the two obvious possibilities are to represent F directly or to represent f. Since techniques for calculating with probability density functions are better known than techniques for calculating with distribution functions, we next turn our attention to representations suitable for density functions.

## Probability Density Functions

The problem of representing a pdf is very much the general problem of approximating a function of a real value. In fact, the problem is complicated by the fact that many pdfs are not even continuous (e.g., uniform distributions), much less differentiable. Given such an approximation problem, the analyst must first choose the "form and norm" [Rice 64] to be used. The "form" refers to the class of approximating functions from which the actual approximation will be chosen; the "norm" refers to the error measure (i.e., least squares, least maximum error, etc.) to be used in selecting a particular function of the chosen form to be the approximation. Once these choices have been made, we can then proceed to comment on the existence and uniqueness of the solution to the approximating problem.

Our criteria for selection of an appropriate form is strongly colored by our usage of these approximations for the f-convolution operations. Thus, our emphasis is significantly different from that in texts on function approximation. We will emphasize the form of our representations and the costs of computing with those forms more than we will discuss norms and the impact of the norms on the choice of a representative from the class of approximating functions. The reason for this choice is simply that the impact of the norms is no different for our problems than for others and has already been extensively considered in the literature. On the other hand, the efficiency and accuracy issues for calculating f-convolutions depend critically on the form of the approximating function and have not been previously considered in the literature in a unified way.

## Representation by Sampling Techniques

The most obvious representation is merely to choose evenly spaced points along the pdf and tabulate their values. Depending upon the techniques used to interpolate values for intermediate points, a range of representations is defined. For example, if the value of a pdf p at a point x is chosen as the value at the nearest tabulated point, then we are approximating p by a mixture of uniform distributions of a constant width.

This class of approximations is interesting because it provides representations which are easy to use to compute f-convolutions. In particular, if we are interested in the base operations of addition and subtraction, then the corresponding f-convolutions are convolution and correlation. The recent work with Fast Fourier Transforms has led to rapid techniques for calculating these results [Stockham 69]. In the next section, we will describe briefly the basic computational technique and in the following section indicate the major error sources.

## Convolution by FFT Techniques

The Fast Fourier Transform algorithms are actually a set of algorithms which can compute the Discrete Fourier Transform of a series of N data points in time $T = kN \log_2 N$. These algorithms became well-known following the publication of a paper by Cooley and Tukey in 1965 [Cooley 65], although the ideas can be traced back to Runge in the early 1900's (see [Cooley 67]). Recent hardware efforts have reduced k to values as small as 500 ns using special-purpose hardware [Allen 75], while values of 60 μs have been reported on machines such as the IBM 7094 [Stockham 66].

As first described by Stockham [Stockham 66], the FFT algorithms can be used to significantly speed the calculation of convolutions and correlations. The technique depends on the fact that the product of the Discrete Fourier Transform (DFT) of any two sequences of points is equal to the DFT of the circular convolution of the two sequences. To obtain ordinary convolution, one must pad the desired points with a sufficient number of zeros so that the circularity is irrelevant. The DFT and inverse DFT can both be calculated by FFT techniques resulting in a significant time savings compared to standard techniques if the number of points involved is medium sized or larger. Stockham's data show the crossover point between N = 24 and N = 32.

The DFT is defined as the discrete analog of the Fourier transform, namely

$$F(t) = \sum_{k=0}^{N-1} f(k)\, w^{kt}, \qquad t = 0, 1, \ldots, N-1, \qquad w = e^{2\pi i/N} \quad .$$

If we are given two sequences of points, f and g (we will assume f and g are both sequences of N points), then their circular convolution is defined as

$$h_c(l) = \sum_{n=0}^{N-1} f(n)\, g((l-n) \bmod N) \quad .$$

It may be shown [Gold 69] that if we calculate the DFTs of f and g, and multiply these and apply the inverse DFT, the result is precisely $h_c$. While this is a complicated way to perform a simple calculation, there is a significant speed advantage which justifies the effort. The direct way of calculating $h_c$ would require N multiplications and $N-1$ additions for each value of l. This gives a total time of $k_c N^2$ where $k_c$ is the time to perform one multiply/add and associated bookkeeping. Calculation of the DFT, however, requires $k_f N \log_2 N$ time as does calculation of the inverse DFT. This yields a total time of $3k_f N(\log_2 N + \lambda)$ which is smaller for large N. (These calculations are for N a power of 2; similar savings hold for other Ns.)

For work with probability distributions, we are not interested in circular convolutions, but rather in "aperiodic" convolutions of the form

$$h(l) = \sum_{n=0}^{N-1} f(n)\, g(l-n) \quad .$$

One way to achieve this effect using the DFT, is to use a larger value of N. That is, if f and g are given at M points, then assume their value is zero outside of that range. Extend the sequences with enough zeros (M is always enough), and perform a circular convolution of these extended sequences. A predictable portion of the resulting circular convolution is then the desired aperiodic convolution (see [Stockham 69] for more details).

<u>Error Analysis</u>

Since we have restricted ourselves to absolutely continuous distribution functions in this section, their pdfs exist almost everywhere. We will further assume that our pdfs are bounded on $(-\infty, \infty)$. Then we may conclude that the convolution of our pdfs exists and is also bounded on $(-\infty, \infty)$ [Apostol 57]. Let us examine now more carefully the implications of calculating convolutions by FFT techniques when we are talking about absolutely continuous distribution functions.

The first thing we must observe is that the use of FFT techniques in the calculation does not change the error characteristics significantly. That is, once we have sampled our two input pdfs f and g at some chosen points, whether we calculate the convolution of these sequences of points by FFT techniques or through direct inner product evaluations, the results will be numerically identical (remember that we made an assumption that arbitrary precision arithmetic is available; in fact, the round-off error propagation of the FFT implementations has been shown by experience to be as good or better than that obtainable by summing products [Stockham 66]. Further, by careful choice of quantization of the probability values, and number of samples, one can take advantage of FFT-related techniques such as the Fermat Number Transform to yield no round-off errors and further improved speeds [Agarwal 75]).

The important errors in the technique arise rather from the sampling process on the input pdfs. Let us try to bound these errors more carefully. If we wish to calculate the convolution of two pdfs f and g, then we wish to evaluate

$$h(x) = \int_{-\infty}^{\infty} f(y)\, g(x-y)\, dy$$

and this exists if f and g are absolutely integrable (true for all pdfs) and bounded on $(-\infty, \infty)$.

The first thing we must do is choose an upper and lower bound for the domain of each of the functions f and g which represent the limits of the sampled versions of these functions. If we choose limits for f only, and these limits are [a, b], resulting in ignoring a weight of $\epsilon_f$ in the tails of f, then in fact we are evaluating

$$\hat{h}_f(x) = \int_{a}^{b} f(y)\, g(x-y)\, dy$$

The error caused by this can be directly calculated

$$h(x) - \hat{h}_f(x) = \int_{-\infty}^{a} f(y)\, g(x-y)\, dy + \int_{b}^{\infty} f(y)\, g(x-y)\, dy \quad .$$

Sinee both integrals are positive (f and g are always positive), we have

$$\int_{-\infty}^{\infty} |h(x) - \hat{h}_f(x)|\, dx = \int_{-\infty}^{\infty} \left[\int_{-\infty}^{a} f(y)\, g(x-y)\, dy + \int_{b}^{\infty} f(y)\, g(x-y)\, dy\right] dx \quad .$$

If $f(y)\, g(x-y)$ is a eontinuous funetion of $x$ and $y$, then we ean interehange the order of integration to obtain

$$\int_{-\infty}^{\infty} |h(x) - \hat{h}_f(x)|\, dx = \int_{-\infty}^{a} f(y) \int_{-\infty}^{\infty} g(x-y)\, dx\, dy + \int_{b}^{\infty} f(y) \int_{-\infty}^{\infty} g(x-y)\, dx\, dy \quad .$$

Since $g$ is a pdf, $\int_{-\infty}^{\infty} g(x-y)\, dx = 1$ for any $y$. So we have

$$\int_{-\infty}^{\infty} |h(x) - \hat{h}_f(x)|\, dy = \int_{-\infty}^{a} f(y)\, dy + \int_{b}^{\infty} f(y)\, dy$$

$$= \epsilon_f \quad .$$

Thus, the error in the eonvolution result that we make by assuming one pdf has no tails is given preeisely by the weight in those tails when the error measure is an $L_1$ norm.*

If we also use a tail-limited version of $g$ (call it $\hat{g}$), then the error we make is similar, that is

$$\int_{-\infty}^{\infty} |\hat{h}_f(x) - \hat{h}_{fg}(x)|\, dx = \epsilon_g$$

where $\epsilon_g$ is the weight in the ignored tails of $g$. Then, by the triangle inequality, the total error between $h(x)$ and $\hat{h}_{fg}(x)$ is bounded by $\epsilon_f + \epsilon_g$, the sum of the weights of the ignored tails.

Once we have chosen an upper and lower bound, we must next choose a sampling interval whieh is the same for $f$ and $g$ if we use FFT techniques. This sampling interval is then used as the sampling interval for $\hat{h}_{fg}$. Thus, we have

$$\hat{h}_{fg}(c + k\Delta t) = \int_{a}^{b} f(y)\, \hat{g}(c + k\Delta t - y)\, dy$$

are the exact values for the sampled and tail-limited convolution we are seeking. Because we are sampling the input with the same $\Delta t$ as the desired output and then calculating the convolution of two sequences of points, we actually calculate

$$\hat{h}^*_{fg}(c + k\Delta t) = \left[\sum_{l=0}^{(b-a)/\Delta t - 1} f(a + l\Delta t)\, \hat{g}(c + k\Delta t - (a + l\Delta t))\right] \Delta t$$

* The $L_n$ family of norms is often used as measures of distanee between funetions. They are defined by:

$$L_n(f, g) = \left[\int_{-\infty}^{\infty} |f(x) - g(x)|^n\, dx\right]^{1/n} \quad .$$

and the error between the discrete sum and the integral is given by

$$\hat{h}_{fg}(c + k\Delta t) - \hat{h}^*_{fg}(c + k\Delta t) = \int_a^b f(y)\,\hat{g}(c + k\Delta t - y)\,dy$$

$$- \Delta t \sum_{l=0}^{(b-a)/\Delta t-1} f(a + l\Delta t)\,\hat{g}(c + k\Delta t - (a + l\Delta t)) \quad .$$

This difference is precisely the difference between the Riemann sum representation of the integral and the integral itself. The absolute value of this error is known [Davis 75] to be $\leqslant (b - a) \times u(\Delta t)$, where

$$u(\Delta t) = \max_{|x_1 - x_2| \leqslant \Delta t} |f(x_1)\,\hat{g}(c + k\Delta t - x_1) - f(x_2)\,\hat{g}(c + k\Delta t - x_2)|$$

when $f(x)\,\hat{g}(c + k\Delta t - x)$ is continuous in $x$. If $\frac{d}{dx}\, f(x)\,\hat{g}(c + k\Delta t - x)$ exists for all $k$ and is bounded, then the error at each point in the resulting convolution reduces proportionally to $\Delta t$. If instead of choosing $f$ and $g$ values at points $a + l\Delta t$, we choose the values at $(a + l\Delta t + \Delta t/2)$, and $w_k(x) = f(x)\,g(c + k\Delta t - x)$ has continuous second derivative, then the error at each $k$ is given by

$$\frac{(b - a)\,(\Delta t)^2}{24}\,\frac{d^2 w_k(\delta_k)}{dx^2}$$

where $a < \delta_k < b$. The total error for the tail-limited convolution is then

$$\sum_{k=1}^{2(b-a)/\Delta t} |\hat{h}_{fg}(c + k\Delta t) - \hat{h}^*_{fg}(c + k\Delta t)| = \sum_{k=1}^{2(b-a)/\Delta t} \left[(b - a)\,(\Delta t)^2\,\frac{d^2 w_k(\delta_k)}{dx^2}\right] \Big/ 24 \quad .$$

If we let

$$\lambda = \max_k \frac{d^2 w_k(\delta_k)}{dx^2}$$

then

$$\sum_{k=1}^{2(b-a)/\Delta t} |\hat{h}_{fg}(c + k\Delta t) - \hat{h}^*_{fg}(c + k\Delta t)| \leqslant [(b - a)^2\,\lambda\Delta t]/12 \quad .$$

Thus, the total error in the result is bounded by the sum of the errors caused by removing the tails, and the errors caused by sampling in the intervals that remain.

$$\int_{-\infty}^{\infty} |h(x) - \hat{h}^*_{fg}(x)|\,dx \leqslant \epsilon_f + \epsilon_g + \frac{(b - a)^2\,\lambda\Delta t}{12} \quad .$$

In order to derive this bound, we needed one assumption in addition to those constraints on the pdfs we had at the beginning of the section, namely that

$$\frac{d^2}{dx^2} f(x)\, g(c + k\Delta t - x)$$

exists and is continuous. This is a fairly strong assumption for probability distribution functions, but I have not been able to obtain any bounds under weaker assumptions.

## Correlation

The preceding development was expressed in terms of convolution. In fact, the computing techniques and error bounds for correlation are basically identical. Recognizing that $X - Y$ is identical to $X + (-Y)$, we need merely look at what is required to represent the negation of a random variable expressed in sampled form. The negation operation implies negating and reversing the upper and lower bounds, and taking the sample points in the reverse order. The sampling interval remains constant. If we go through the error bounds section consistently replacing $g(c + k\Delta t - x)$ with $g(c + k\Delta t + x)$, then the error bounds themselves are not affected and the derived results all apply to correlation, when computed in this fashion.

## *-Convolution

When the desired base operation is multiplication, we can also try to take advantage of FFT techniques to perform the calculation. To do this, we note the identity

$$Y * Z = \exp(\ln Y + \ln Z) \quad .$$

Thus, by performing three unary operations, i.e., operations on one distribution, and a +-convolution, we obtain a *-convolution. Note that if $X = \ln Y$, then we have (assuming $Y > 0$ to avoid difficulties and also for some $k$, $\int_0^\infty t^k f_Y(t)\,dt$ exists and $Z$ meets the same conditions)

$$f_X(x) = e^x f_Y(e^x)$$

[Parzen 60].

If we calculate the Fourier transform of $f_X$ in terms of $f_Y$, we obtain the following expression:

$$f_X^*(s) = \int_{-\infty}^{\infty} f_X(x)\, e^{isx}\, dx$$

$$= \int_{-\infty}^{\infty} e^x f_Y(e^x)\, e^{isx}\, dx$$

$$= \int_{-\infty}^{\infty} e^{x(is+1)}\, f_Y(e^x)\, dx$$

let $t = e^x$, we have

$$= \int_0^{\infty} t^{(is+1)}\, f_Y(t)\, \frac{dt}{t}$$

$$f_X^*(s) = \int_0^{\infty} t^{is} f_Y(t)\, dt \quad .$$

Checking in our tables [Bateman 54] we find that, with a change of variables $p = 1 - is$, we have

$$f_Y^M(p) = f_X^*(1 - is)$$

where $f_Y^M$ is the Mellin transform of $f_Y$. Thus, if we take the Mellin transform of the distributions of our two random variables, multiply these transformed distributions and calculate the inverse transform, we find that we have performed a $*$-convolution (see [Ditkin 65] for more on the properties of the Mellin transform).

This, of course, is only useful if these transforms can be performed efficiently, using FFT-type techniques. Of course, we have just seen the necessary trick, i.e., calculate the pdfs corresponding to ln Y and ln Z and take their Fourier transform. However, the accuracy begins to degrade because a resampling must occur to keep the sample points evenly spaced both before and after the ln operation. I do not know if FFT-type techniques can be directly employed on the original pdf to calculate its Mellin transform and its inverse directly.

## Other f-Convolution Operations

In general, if h-convolution for an arbitrary base function h is desired, we have the following equation:

$$F(t) = \underset{\substack{(x_1 \ldots x_n) \\ \ni h(x_1 \ldots x_n) \leqslant t}}{\int \cdots \int} f(x_1 \ldots x_n)\, dx_1 \ldots dx_n$$

where f is the point distribution function of $X_1 \ldots X_n$. This multiple integral can be directly attacked by numerical integration techniques, if necessary (see [Stroud 71] for a survey of techniques). These, however, are likely to be more expensive and less accurate in computation than schemes tuned to a particular f-convolution.

## Other pdf Representations

There are a large number of other ways to represent pdfs in addition to the sampling techniques described above. These divide into two major classes: (1) representation by functions of a special class, and (2) representation by series expansion. We discuss below an example of each type. These particular choices are well known in the statistical literature, but many examples of each type can be found, each with its own advantages and disadvantages.

One well-known special class of pdfs which can assume a broad variety of shapes and is thus useful for approximation is the Pearson family of pdfs [Kendall 63]. These pdfs can all be characterized by the following differential equation, where f is the pdf

$$\frac{df}{dx} = \frac{(x - a)\,f}{b_0 + b_1 x + b_2 x^2} \quad .$$

It is simple to derive from this equation a number of properties of f.

(1) df/dx vanishes at the point x = a, and only at that point. Thus these distributions are uni-modal, but in special cases. they may be J-shaped or U-shaped.

(2) There is (except for special cases) smooth contact with the x-axis at the extremities, so that df/dx vanishes when f = 0.

(3) The parameters a, $b_0$, $b_1$, and $b_2$ can all be determined from the first four moments.

(4) The Normal, Beta, Chi-square, Student's t–, and Gamma pdfs are all special cases of the Pearson family.

Since a Pearson pdf can be determined from its first four moments, the relations between the defining parameters and the first four moments can be used to move from one to the other. That is, given the first four moments, one can determine the parameters a, $b_0$, $b_1$, and $b_2$ defining the Pearson pdf, and conversely, given a, $b_0$, $b_1$, and $b_2$, one can calculate the first four moments. Indeed, given a Pearson pdf, there is a recurrence relation for the moments which will calculate moments of higher orders easily, but the first four determine all of them. If two Pearson pdfs f and g are convolved to form a pdf h, h is not, in general Pearson, but a Pearson representative which matches it in the first four moments can be determined. (This can be demonstrated by comparing the higher order moments ($\mu_5$, $\mu_6$, etc.) of the Pearson representative for h with the values for the true h for some numerical examples.)

Generally, there is no advantage to using the Pearson family in place of representation by the first four moments (see discussion of representation by moments, below). The moments are easier to compute with for f-convolutions and easier to obtain in the first place.

Another type of representation for pdfs is in terms of the coefficients in a series of orthogonal polynomials. One of the best-known of these schemes is the Gram-Charlier Type A series [Kendall 63]. This is an expansion in terms of Tchebycheff-Hermite polynomials.

The major properties of the Gram-Charlier series for use in a statistical compiler are:

(1) The coefficients are easily calculated from the moments, and the moments may be easily calculated from the coefficients,

(2) It is easy to convert from a pdf in this form to a cdf in this form, and conversely,

(3) The sum of a finite number of terms of the series may produce negative values for the probabilities near the tails,

(4) A series of n terms may be a poorer fit than a series of n-1 terms, and

(5) If there is a significant skew in the pdf, then the fit will probably be poor.

f-convolutions would be performed on the pdfs by manipulating the coefficients in the Gram-Charlier representation of the pdfs. Closed-form relations between the coefficients can be obtained for cases when the relations between the moments are known: for the case of +-convolution, these relations are particularly simple. If our representation is in terms of the Gram-Charlier series about the mean, and the representation for X is given by $<\bar{x}, x_0, x_1, \ldots>$, where $\bar{x}$ is the mean of the random variable X, and $x_i$ is the coefficient of the $i^{th}$ term in the series expansion, then for $Z = X + Y$, (X and Y independent) we have the following relations:

$$\bar{z} = \bar{x} + \bar{y}$$

$$z_0 = 1$$

$$z_1 = 0$$

$$z_2 = x_2 + y_2 + 1/2$$

$$z_3 = x_3 + y_3$$

$$z_4 = x_4 + y_4 + x_2 y_2 + 1/8$$

$$\vdots$$

This is obtained from combining the expressions for the Gram-Charlier coefficients ([Kendall 63, p. 157]) with expressions for the moments of $Z$ in terms of the moments of $X$ and $Y$.

For $*$-convolution, the same approach can be applied to obtain:

$$\bar{z} = \bar{x}\bar{y}$$

$$z_0 = 1$$

$$z_1 = 0$$

$$z_2 = 2x_2y_2 + x_2 + y_2 + 1/2$$

$$z_3 = 6x_3y_3$$

$$z_4 = 24x_4y_4 + 12x_4y_2 + 12x_2y_4 + 10x_2y_2 + 3x_4 + 3y_4 + x_2 + y_2 + 1/4$$

$$\vdots$$

These expressions are relatively easy to deal with, although the number of terms begins to get unwieldy for $*$-convolution with higher order series.

Given the basic accuracy limitations on Gram-Charlier series outlined above, they can only be recommended where the ease of conversion to cdf representations is important, and "nice" pdfs are used. Usually, one of the other forms will be more convenient for computation and avoid the accuracy limitations inherent in Gram-Charlier.

## Representation by cdf

In order to obtain error bounds in the discussion of convolution above, we had to make assumptions which are hard to justify regarding the smoothness and differentiability of our pdfs. Generally, trying to represent and compute with functions which are not smooth is subject to many pitfalls and should be avoided where possible. For this reason, an attractive alternate representation is the distribution function, also referred to as the cumulative distribution function (cdf) to emphasize the distinction from the pdf more strongly.

The choices for representation of the cdf are similar to the choices for representing pdfs, ranging from sampling to series and continued-fraction expansions. We discuss below what some operations on random variables imply about operations on the corresponding cdf and leave the implementor to choose the most convenient representation for the expected cdfs.

The first thing to note is that the calculation of the $+$-convolution for two random variables cannot be expressed directly in terms of the cdfs of those random variables.

$$F_Z(z) = P[X + Y \leqslant z]$$

$$= \int_{-\infty}^{\infty} \int_{-\infty}^{z-x} f_{X,Y}(x, y)\, dy\, dx \quad .$$

Assuming X and Y are independent, we have

$$F_Z(z) = \int_{-\infty}^{\infty} f_X(x) \int_{-\infty}^{z-x} f_Y(y)\, dy\, dx$$

$$= \int_{-\infty}^{\infty} f_X(x)\, F_Y(z - x)\, dx$$

Computing this requires calculating $f_X$ and then forming its convolution with $F_Y$, but I know of no way to avoid the calculation of the pdf of one of the random variables.

However, there are other operations among random variables which translate quite directly into operations on their cdfs. For example, consider $Z = \max(X, Y)$.

$$F_Z(z) = P[\max(X, Y) \leqslant z]$$

$$= P[(X \leqslant z) \text{ and } (Y \leqslant z)] \quad .$$

Assuming X and Y are independent, we have

$$F_Z(z) = F_X(z)\, F_Y(z) \quad .$$

Similarly, for $Z = \min(X, Y)$, we obtain

$$F_Z(z) = F_X(z) + F_Y(z) - F_X(z)\, F_Y(z) \quad .$$

Note that performing max-convolution with pdfs would require calculating both cdfs.

Another operation which is particularly convenient with cdfs is the following (expressed in ECL):

```
CHOICE <- EXPR(X: real\random,
               Y: real\random,
               Z: real\random,
               k: REAL;
               real\random)
          [) X LE k => Y; Z (];

W <- CHOICE(X, Y, Z, k);
```

Then we have assuming independence of our random variables,

$$F_W(w) = P[((X \leqslant k) \text{ and } (Y \leqslant w)) \text{ OR } ((X > k) \text{ and } (Z \leqslant w))]$$

$$= F_X(k)\, F_Y(w) + (1 - F_X(k))\, F_Z(w) \quad .$$

Generally, when there are many conditional operations to be performed, especially those involving comparison with REALs, the cdf is the more convenient form to use.

## Moments

Statisticians have long used moments to describe succinctly the characteristics of a distribution. Such measures as variance, skewness, and kurtosis are related to the moments about the origin or about the mean in direct and simple ways. Moments have the additional advantage from our viewpoint of being particularly convenient for certain f-convolution operations. In many cases, the answer desired by the poser of the original problem is expressed directly in terms of moment measures, i.e., what is the mean and standard deviation of X? We must be wary, however, because not all distributions have moments of all orders and for certain cases, no moments at all exist. In the following, we will assume that the input distributions we use have moments of all the degrees that we use to represent the distributions.

An f-convolution which is particularly simple to perform with a moment representation is *-convolution. For, if $Z = X * Y$, we have the following

$$E[Z^n] = E[(XY)^n]$$

$$= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} (xy)^n F_{X,Y}(x,y)\,dx\,dy \quad .$$

Assuming X and Y are independent, we have

$$E[Z^n] = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} x^n y^n f_X(x)\, f_Y(y)\,dx\,dy$$

$$= \int_{-\infty}^{\infty} y^n f_Y(y) \int_{-\infty}^{\infty} x^n f_X(x)\,dx\,dy$$

$$= \int_{-\infty}^{\infty} y^n f_Y(y)\, E[X^n]\,dy$$

$$= E[X^n]\,E[Y^n] \quad .$$

Thus, if X and Y have moments to degree m, then so will Z. In words, the moments of a product of independent random variables are the product of the moments of those random variables.

Equally simple is the form $Z = X^k$ for k a positive integer. Here we have directly that $E[Z^n] = E[X^{kn}]$. In this case, Z will have moments to degree m if X has moments to degree km.

Since $E[g_1(X) + g_2(X)] = E[g_1(X)] + E[g_2(X)]$, we can extend these results to obtain, if

$$Z = \sum_i \sum_j \sum_k a_{ijk} X^i Y^j W^k$$

then we have, assuming independence of X, Y, and W,

$$E[Z] = \sum_i \sum_j \sum_k a_{ijk} E[X^i]\, E[Y^j]\, E[W^k] \quad .$$

To obtain the higher moments, expand $Z^n$ symbolically to obtain

$$Z^n = \sum_i \sum_j \sum_k b_{ijk} X^i Y^j W^k \quad .$$

Then,

$$E[Z^n] = \sum_i \sum_j \sum_k b_{ijk} E[X^i] \; E[Y^j] \; E[W^k] \quad .$$

Thus, we can evaluate the moments of a multinomial directly from the moments of its input distributions, assuming independence of those input distributions and existence of moments of adequately high degree. In the above example, if the multinomial for Z is of degree k in X, then evaluation of $E[Z^n]$ requires the existence of moments of degree up to kn for X.

## Characteristic Functions

The characteristic function is the continuous Fourier transform of the pdf; it is also the moment generating function. Whenever the pdf exists, so does the characteristic function, although an arbitrary function is not in general the characteristic function for some random variable. The characteristic function has some convenient properties for f-convolution, namely if $\lambda_X$ and $\lambda_Y$ are the characteristic functions for the random variables X and Y, then $\lambda_{X+Y} = \lambda_X \lambda_Y$ is the characteristic function for their sum, assuming independence.

Note that the characteristic function representation was used to perform convolution for sampled pdfs. That is, to perform convolution, we first took the DFT of the input pdfs to obtain an approximation to the characteristic function and then used the property indicated above to perform the convolution. If a sequence of +-convolutions are to be performed, then it is more efficient overall to leave the pdfs in the characteristic function form and only convert back to sampled form after the entire sequence.

## Atomic Distributions

We say a distribution F is atomic if F is concentrated on its set of atoms, i.e., if the atoms of F contain the complete mass of the distribution. It is easy to show that there are at most denumerably many atoms for any atomic distribution.

Just as in the absolutely continuous case, there exists an associated function, the probability mass function (pmf) p, such that

$$F(x) = \sum_{a_i \leqslant x} p(a_i)$$

where $a_i$ ranges over the set of atoms for F which are less than or equal to x.

The pmf then maps the atoms $a_i$ of F into the probability mass associated with those atoms, i.e., the amount of increase of F at $a_i$.

The representation choices for atomic distributions are similar in many ways to the choices for absolutely continuous distributions. For representations such as cdf or moments, the discussions already presented apply equally well here.

However, for other representations, such as pmf or characteristic functions, some modications are necessary. Also there are some important subclasses of the atomic distributions, where some specific techniques can be employed. These subclasses are the finite atomic distributions (those with a finite number of atoms), and the integer distributions (all atoms are integers).

We discuss below some of the effects of these differences.

## Representation by pmf

The pmf may be compared with the pdf representations described earlier for absolutely continuous distributions. The pmf by its nature is in a sampled form, but there is no guarantee that the spacing is uniform, or that only a finite set of samples can provide a complete representation.

Moreover, each atom represents only its own point in pmf, whereas in a sampled pdf each sample point represents an interval. This effect is most striking in looking at unary operations: e.g., consider the following theorem [Parzen 60].

> If $y = g(x)$ is differentiable for all $x$, and either $g'(x) > 0$ for all $x$ or $g'(x) < 0$ for all $x$, and if $X$ is a continuous random variable, then $Y = g(X)$ is a continuous random variable with probability density function given by
>
> $$f_Y(y) = f_X(g^{-1}(y)) \left| \frac{d}{dy} g^{-1}(y) \right| \text{ if } y \epsilon \text{ range of } g$$
> $$= 0 \text{ otherwise.}$$

For contrast, the relationship between pmfs is quite different, namely

$$f_Y(y) = f_X(g^{-1}(y)) \text{ if } y \epsilon \text{ range of } g$$
$$= 0 \text{ otherwise}$$

since we are only interested in the mapping of individual atoms.

If there are too many atoms in the distribution to store them individually, then some grouping of atoms must be performed. The effects of this grouping on f-convolution operations will depend on the nature of the distribution and the spacing between the atoms. If the atoms are "dense enough," then the distribution may be approximated as a continuous distribution, however, the exact specifications for "dense enough" will depend on the application.

Although the number of atoms in the input distributions may be small, this does not preclude the possibility of generating large intermediate results. For example, if the distributions for $X$ and $Y$ contain $n$ atoms each, then the distribution for $X + Y$ will contain anywhere from $2n$ to $n^2$ atoms depending on the spacing of the atoms in $X$ and $Y$.

Thus, it may be necessary to dynamically alter the grouping of atoms into sample points in order to maintain a feasible number of points in intermediate distributions, although again the error effects are difficult to predict.

## Integer Distributions

The case of integer distributions is of particular practical importance. Much of the discussion under probability mass functions also applies to integer distributions, but there is an additional technique which is of particular interest in this case, namely representation by generating functions.

The basic concept is to represent the probability values as the coefficients in a polynomial. The entire polynomial then becomes the representation for the distribution, or its generating function.

That is, let

$$A(s) = p(0) + p(1)\,s + p(2)\,s^2 + \ldots$$

then for pmfs, A(s) converges absolutely for $|s| \leq 1$.

Some of the properties of generating functions make them particularly convenient for f-convolution operations. For example, if A(s) and B(s) are the generating function representation for the random variables X and Y, then we have $A(s) * B(s)$ as the generating function representation for X + Y.

However, the most interesting application of generating functions in the statistical compiler area is for evaluating the following function of random variables, expressed as an ECL program.

```
SUM   ←   EXPR (X:  real\random,
                N:  real\random;
                    real\random)
          [] DECL Z:  real\random BYVAL point (0);
          /* ' Z is initialized to a distribution whose only atom is 0 ';
             FOR i FROM 1 TO N
             REPEAT
               DECL S:  real\random IID X;
               /* ' S is a series of independent distributions all with
               identical distribution functions which are the same as the
               distribution function for X ';
               Z ← Z + S;
               END;
            Z;
          [];
```

That is, SUM calculates the distribution of the sum of N values independently chosen from the distribution X, where N is a random variable. The generating function representation of this is surprisingly simple [Feller 57, Vol. 1, p. 268]: if A(s) is the generating function for N, and B(s) is the generating function for X, then A(B(s)) is the generating function for SUM(X, N).

Remember, however, the difficulties imposed by the addition of looping capabilities to our set of base functions. We are able to avoid the problem only because the restrictions for practical computations are so severe. If either X or N is represented as an infinite sequence of coefficients of a generating function, then the computation will never terminate. It concludes only if the generating functions for both X and N can be expressed in closed form, i.e., if X is a Poisson distribution with $\lambda$, then its generating function is $e^{-\lambda+\lambda s}$. Thus, although this is a convenient way to represent the work to be performed, it still requires a potentially infinite amount of computation.

## Conversion Between Representations

Since it is probable that the implementor of a statistical compiler will wish to provide a broad variety of base functions, and since the work required to perform a particular f-convolution varies dramatically with representation, it is sometimes desirable to change the representation during the computational process. In some cases, these conversions are simple to perform numerically and can be done with high accuracy. However, some conversions have theoretical bounds on their accuracy and these bounds may be large in practical cases. The purpose of this section is to indicate the nature of the conversions between the types of representations discussed above.

### From pdf to cdf

Since the cdf is the integral of the pdf, and integration is a smoothing operation, the results can be expected to be satisfactory. Depending upon the representations used for the pdf and cdf, various numerical integration techniques apply. See [Davis 75] for a survey of techniques which could be used. For certain representations, e.g., the Gram-Charlier series, the conversion can be simply performed in terms of the parameters describing the series.

### From pdf to Moments

If the representation of the pdf is as a Pearson distribution or a Gram-Charlier series, theoretical expressions for the moments can be obtained directly from the parameters of the representation which are convenient for calculation.

When a sampling representation of the pdf is used, the obvious calculation for the moments is to evaluate

$$\hat{\mu}_m' = \sum_{k=1}^{(b-a)/\Delta t} (a + k\Delta t)^m f(a + k\Delta t - \Delta t/2) \quad .$$

Although this inner product can be directly evaluated with appropriate numerical care ([Wilkinson 63]), there remain a number of practical problems. For example, if the unsampled sections in the tails of the distribution have enough weight, then any moment calculated from the sampled version will be unreliable. Indeed, the moments calculated from the sampled distribution always exist, whereas the moments of the infinite range distribution from which it is derived do not necessarily exist. Further, even in cases where this is not an issue, we are still making the assumption that the weight in an interval $\Delta t$ is concentrated at the center of the interval.

These problems are the same as those arising when one attempts to estimate the moments of a distribution from a sample selected from the population described by that distribution, and that sample has been grouped. If the distribution is quite well behaved (to be specified shortly), then this calculation can be accurately performed, with Sheppard's corrections used to correct for the grouping effect [Kendall 63]. The conditions for application of Sheppard's corrections are, however, fairly stringent.

In order to obtain a reliable value for the moment $\mu_m'$, we must have a number of conditions fulfilled. (1) If $y = x^m f(x)$, then the first $2m - 3$ derivatives of $y$ should approximately vanish at the ends of the sampled interval (i.e., f must have high-order contact with the axis in its tails), (2) $y$ must have $2m$ derivatives, and (3) $4y^{(2m)}(\epsilon)(b - a)\Delta t^{2m-1}/(2\pi)^{2m}$ for some $\epsilon$ in the interval [a, b] provides a measure of the error in the calculation, and should be small.

### From pdf to Characteristic Function

The calculation of the characteristic function from the pdf is merely the calculation of the continuous Fourier transform of the pdf. Fast Fourier transform techniques can be used to convert rapidly between sampled representations of the pdf and the characteristic function, and conversely. The errors caused by converting a continuous curve to a sampled version, and then calculating a discrete Fourier transform and using it as an approximation to the continuous Fourier transform are well known and discussed in several articles and texts (e.g., [Cooley 70]).

Generally, this conversion will work satisfactorily if (1) the pdf does not have very "high-frequency" components, and (2) the characteristic function is desired for relatively small values of its argument. If we sample the pdf at interval $\Delta t$, then the characteristic function obtained by FFT techniques can be in error by 100 percent when its argument is $1/(2\Delta t)$ (Nyquist frequency) because of sampling. This "aliasing" error will be acceptably small only if the value of the characteristic function is negligible at this point. In addition, chopping off the tails of the pdf, which is a multiplication in the original domain, represents a convolution operation in the transformed domain. The function convolved with is of the form $(\sin x)/x$, producing significant errors in the characteristic function referred to as "leakage" errors. These "leakage" errors can be compensated for by using a "window" function on the original pdf which is not a rectangular data window, but rather some function with more tapered ends. At this stage, the selection of an appropriate window is very much an art and is discussed in [Bergland 69].

## From Moments to cdf

The problem of conversion from moments to the cdf and pdf forms of representation is a classical mathematical problem, and has been attacked by such mathematicians as Tchebycheff, Markoff, Stieltjes, Hausdorff, and Hamburger. The "Stieltjes integral" was introduced in 1894 in a paper which also provides a solution to one variant of the problem of moments.

This wealth of mathematical interest derives from the fact that the problem is hard enough to be interesting, yet fruitful enough to produce many fascinating results. From our viewpoint, however, these results are essentially negative, and we examine their relation to our problem below.

The first variant of the moment problem, referred to as the Hamburger moment problem, assumes we are given an infinite sequence of numbers and asked whether these are the moments for some distribution function on the infinite interval $(-\infty, \infty)$, and whether we can find that distribution function. To state that more carefully, we offer the following theorem:

**Theorem:** (see [Shohat 43, Theorem 1.2])

In order that a Hamburger moment problem

$$x_n = \int_{-\infty}^{\infty} t^n dF \quad , \quad n = 0, 1, 2, \ldots$$

shall have a solution, a distribution function F, it is necessary that

$$\Delta_n = \begin{vmatrix} x_0 & x_1 & \cdots & x_n \\ x_1 & x_2 & \cdots & x_{n+1} \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ x_n & x_{n+1} & & x_{2n} \end{vmatrix} \geq 0 \quad , \quad n = 0, 1, 2, \ldots$$

In order that there exist a solution which is an atomic distribution whose set of atoms consists of exactly $(k + 1)$ distinct points, it is necessary and sufficient that

$$\Delta_0 > 0, \quad \Delta_1 > 0, \quad \ldots, \quad \Delta_k > 0, \quad \Delta_{k+1} = \Delta_{k+2} = \quad \ldots \quad = 0 \quad .$$

The moment problem is determined in this case.

The condition stated here is expressed in terms of an infinite sequence of moments, and further, even if true, does not guarantee that the function F is unique. Indeed, counter-examples can be found, e.g., [Kendall 63], since

$$\int_0^\infty x^n \exp(-ax^\lambda) \sin(bx^\lambda)\, dx = 0 \qquad a \geqslant 0, \qquad 0 < \lambda < 1/2$$

the pdfs

$$f(x) = k \exp(-ax^\lambda)\, [1 + \epsilon \sin(bx^\lambda)], \qquad 0 \leqslant x \leqslant \infty, \qquad a > 0, \qquad 0 < \lambda < 1/2, \qquad |\epsilon| < 1$$

have the same infinite set of moments for all $\epsilon$ in the allowed range.

An additional theorem gives the condition for a substantially unique distribution with a given set of moments.

**Theorem:** [Shohat 43, Theorem 1.10]

A sufficient condition that the Hamburger moment problem be determined (i.e., have a "substantially unique" solution) is that

$$\sum_{n=1}^{\infty} x_{2n}^{-1/2n} = \infty \quad .$$

Again this is a condition on an infinite sequence of moments although we have only a finite sequence and the numerical evaluation of such a test is prone to significant errors.

If we restrict ourselves to a finite interval, say [0, 1], and consider the reduced moment problem (i.e., only a finite number of moments are given), the situation is only partially improved. If we are given a sequence of moments $x_0, x_1, \ldots, x_n$, then we can determine effectively two values for $x_{n+1}$ which represent the upper and lower bounds for the next moment. We can further determine atomic distributions with about $n/2$ atoms which achieve both the upper and lower bounds on $x_{n+1}$ (call these $F_u$ and $F_l$). If $D|x$ represents the set of cdfs having the moments $x_0, x_1, \ldots, x_n$, then we have several theorems [Karlin 53].

Definition: Let $D_a|x$ represent the subset of $D|x$ consisting of atomic distributions only.

**Theorem:** The sets $D|x$ and $D_a|x$ are convex.

**Theorem:** The extreme points of $D_a|x$ are those functions F with the number of distinct atoms $\leqslant n + 1$.

**Theorem:** $D_a|x$ is spanned by its extreme points, and in the weak * topology (i.e., using a particular measure of distance between cdfs), $D|x$ is also spanned by the extreme points of $D_a|x$.

**Theorem:** If $F \in D|x$, then the differences $F - F_u$, and $F - F_l$, if not identically zero, each have exactly $n - 1$ sign changes in the interval [0, 1].

An example should help clarify the situation. Consider the Beta distribution with parameters 12 and 6.

$$dF = \frac{1}{B(12,6)} x^{11}(1-x)^5 \, dx \qquad 0 \leqslant x \leqslant 1 \quad .$$

The moments for this distribution may be easily calculated by integration, using the properties of the complete Beta integral to obtain the following values:

$$\mu_1' = 0.6666{,}667$$

$$\mu_2' = 0.4561{,}404$$

$$\mu_3' = 0.3192{,}982$$

$$\mu_4' = 0.2280{,}702 \quad .$$

Following [Karlin 53] we can then determine $F_u$ and $F_l$ to have the same first four moments. The work required to calculate these, given n moments, is approximately the cost of evaluating n determinants of size n/2, finding all the roots of two polynomials of degree n/2 (roots are guaranteed to be real, and in [0, 1]), and finally solving two sets of linear equations, each of size n/2. Performing the calculations we obtain the following: $F_l$ has atoms at 0.0, 0.5612,724, and 0.7720,849 with jumps of 0.0013,5098,5, 0.4951,089, and 0.5035,401, respectively. $F_u$ has atoms at 0.5104,210, 0.7276,077, and 1.0, with jumps of 0.3010,400, 0.6826,570, and 0.0163,031, respectively. The reader may readily verify that these distributions have the same first four moments as those given above. The situation is graphed in Fig. 5-2 (values for the Beta distribution were obtained from [Pearson 68]). As can be seen, the functions are not close approximations to each other, and the pdfs would be even more disparate. The intertwining behavior seen in the graphs is typical, and the theorem above states that any other cdf with the same first four moments must also cross every horizontal and vertical line in the graphs of $F_u$ and $F_l$.

Thus, while we can bound the behavior of cdfs from descriptions of their moments, these bounds are necessarily gross and often in an inconvenient form for further computing. Several authors (e.g., [Burr 42]) have suggested techniques for directly fitting cdfs to a given set of moments, but these techniques have depended on first choosing a family of cdfs and then determining the parameters of the specific curve in that family. Because of the large errors which may occur, these techniques cannot be recommended for use in a statistical compiler.

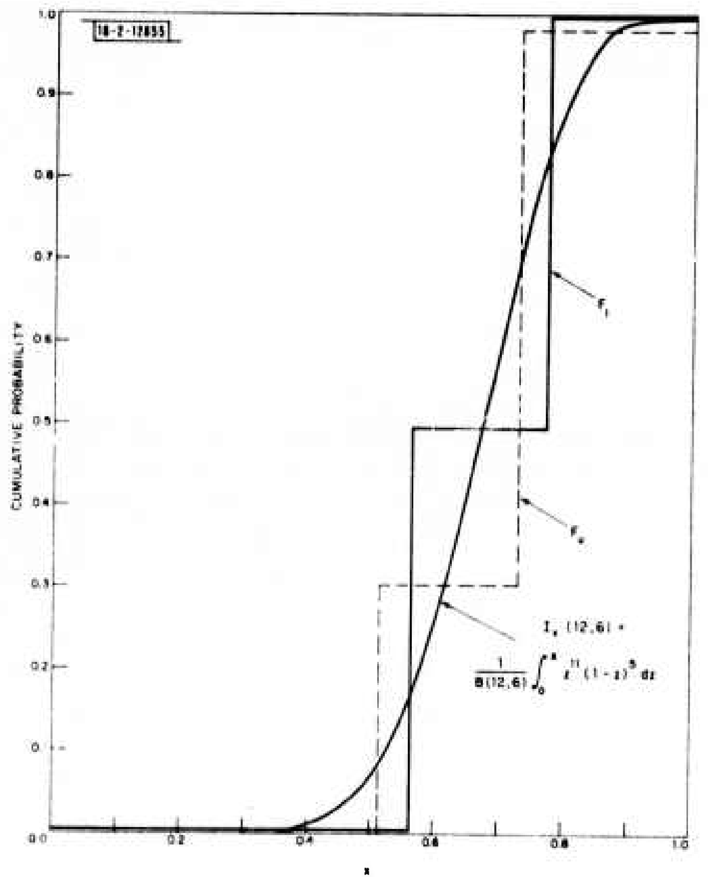

Fig. V-2. Comparison of $F_u$, $F_1$ and Beta distribution with identical $\mu_1'$, $\mu_2'$, $\mu_3'$, and $\mu_4'$.

## VI. DESCRIPTION OF AN IMPLEMENTATION

We now alter the approach of the last few sections and describe a practical, although limited, implementation of a statistical compiler. We will actually describe the system, called STAT, twice; once from the viewpoint of a user, and then again from the viewpoint of the implementor of such a system. STAT is operational on the Harvard PDP-10, in conjunction with the ECL programming system [Harvard 74]. The user of STAT is expected to be conversant with ECL and to be operating interactively.

STAT provides its user with an additional mode ("a new data type"), called *real\random*, and the facilities to create and use objects of this new mode. These facilities are integrated with the ECL system, so that the capabilities to loop, call functions, perform I/O, and other such capabilities of a general system remain available. However, the operations which may be performed on objects of mode *real\random* are only a restricted subset of the ECL operators.

### User Description of STAT

In order to use the STAT system, the user begins by invoking the ECL system in the usual fashion and then "LOAD"s the STAT files. When this operation is completed, the user has available all the usual ECL capabilities and may use them with no modification. In addition, the operator "^" for exponentiation has been provided which may be applied to INTs and REALs with the expected results.

The STAT environment also provides a new mode called *real\random* and several functions ("constructors") to create objects with this mode. In addition, the operators "+", "-", "*", "/", and "^" have been extended to deal with objects of mode *real\random*, although not all possible combinations are allowed.

The new constructors generate random variables with some standard distributions, as well as allowing the user to enter an arbitrary distribution. Although the current version of STAT provides relatively few constructors, it would be easy to add more as no other parts of the system are dependent on the set of constructors provided. The current set of constructors is:

*point* EXPR( *a*:ARITH; *real\random*)

This function produces a random variable which will assume the value *a* with probability 1.

*distribution* EXPR( *tab*:SEQ(VECTOR(2, REAL)); *real\random*)

Returns a random variable whose probability mass function is given in *tab*. *tab* is a sequence of pairs, the first element of which is a probability and the second is a discrete value assumed by the random variable.

*uniform* EXPR( *a*:ARITH, *b*:ARITH; *real\random*)

Returns a random variable whose probability density function is a uniform distribution with lower limit *a* and upper limit *b*.

*gaussian* EXPR ( *mean*:ARITH, *sigma*:ARITH; *real\random*)

Returns a random variable whose probability density function is a normal (Gaussian) centered at *mean*, with standard deviation *sigma*.

*poisson*                    EXPR( *lambda:ARITH; real\random*)

Returns a random variable whose probability density function is a Poisson distribution with mean and standard deviation *lambda*.

The concept of independence of objects of mode *real\random* is critical to the implementation. The constructors described above produce independent random variables each time they are invoked. Variables of mode *real\random* which are calculated through the use of operations are dependent on their computational ancestors, and through them dependent on any of their descendants. An example will help clarify the meaning and necessity for maintaining these dependency relations. Consider the following two pieces of code:

```
1.  a <- uniform(2, 4);  a + a;
2.  a <- uniform(2, 4);  b <- uniform(2, 4);  a+b;
```

The STAT system views these as representing two quite different situations. The first represents choosing a value for the random variable *a* from a uniform distribution, and then adding that value to itself. The resulting distribution is identical to *2*uniform(2,4)*. The second example represents the summing of two independent choices from the identical uniform distribution. *a* and *b* are independent and identically distributed random variables, and the result of their sum is a random variable whose distribution is the convolution of *uniform(2,4)* with itself. The means for the two expressions are the same, but the variance of the first is larger (in fact, twice as large as that of the second). This may not at first be obvious, but can be explained as follows: In the first case, a value is chosen from the distribution and doubled. If it is near the high end, there is no opportunity for it to be countered by a value from the small end. In the second case, if the value selected for *a* is high, there is still a chance that the value selected for *b* will be small, thus reducing the variance of the result.

Example Session 1

```
->a <- uniform(2, 4);
->a + a$
mean = 6.0 var = 1.3333335 skewness = 0.0 kurtosis = 1.8000408
->a <- uniform(2, 4);
->b <- uniform(2, 4);
->a+b$
mean = 6.0 var = 6.6666678E-1 skewness = -1.4016083E-5 kurtosis = 2.4000297
```

STAT extends the usual ECL operators "+", "-", "*", and "/" to apply to *real\random* objects: "+", "-", and "*" may have their left and right arguments be *real\random*, INT or REAL in any combination. "/" permits the left argument (numerator) to be *real\random*, INT, or REAL but allows only INT or REAL for the right argument (denominator). Exponentiation is also provided ("^"); its left argument (base) may be either INT, REAL, or *real\random*, while its right argument (exponent) may be only INT or REAL. If the base is a *real\random*, then the exponent is restricted to a positive integer. All these operators will produce a result of mode *real\random* if either input argument (or both) is of mode *real\random*.

The only output mechanism provided is the extension of the ECL PRINT routine to handle *real\random* objects. Its output is the mean, variance, skewness, and kurtosis [1] of the distribution. This reflects the fact that the internal representation of the random variables is in terms of their moments, and these are the most convenient values available.

These capabilities can be combined to produce the solution to some of the problems we presented in Sec. 1 to motivate this work. For example, consider the problem of locating a high-speed turnoff for a runway. We may write a function of REAL variables, which we call *Distto60MPH*, which will calculate the runway location when the plane reaches 60 MPH. This function will accept three arguments: the touchdown velocity of the aircraft *vt*, the touchdown location *tl* (i.e., distance along the runway), and the deceleration rate *d* (assumed constant). The code for this function, assuming *vt* is in knots, *tl* is in feet and *d* is in ft/sec/sec, is the following:

```
Distto60MPH <-
  EXPR(vt:REAL, tl:REAL, d:REAL; REAL)
    BEGIN
      DECL kl:REAL BYVAL 6.076E3 / 3.6E3;
      DECL vf:REAL BYVAL 6.0E1 * 5.28E3 / 3.6E3;
      ( (vt * kl) ^ 2 - vf ^ 2) / (2.0 * d) + tl;
    END;
```

The constants *kl* and *vf* represent the conversion factor for knots to ft/sec, and the value of 60 MPH in ft/sec, respectively. The result of the function is in feet. This routine can be directly applied to calculate values for specific inputs. For example,

```
->Distto60MPH(113.0, 1500.0, 5.0)$
4.3629695E3
```

This then tells us how far down the runway an aircraft would be when it reached 60 MPH, assuming it had touched down at 1500 ft from threshold, at a velocity of 113 knots, and decelerated at a constant rate of 5 ft/sec/sec.

The values used for the variables are, of course, not known precisely in advance for every aircraft using the runway. However, the distribution of approach speeds for aircraft of specific types is a measurable quantity, and some estimate of the distribution can be obtained from a priori knowledge of the airplane characteristics. The touchdown location distribution can be directly measured by the thickness of tire rubber left by aircraft during their initial touchdown. We will assume that these have been measured or estimated and values chosen from a uniform distribution can be used to approximate them both. In the case of touchdown velocity, values in the range [98,128] knots are a reasonable approximation to the characteristics of medium-scale commercial aircraft (i.e., Boeing 727 class) [Dolat 73], and a plausible range of touchdown location values is [1000, 2000] feet.

With these values, a variant of the program is needed to deal with random variables as inputs. This may be generated by modifying the above program to change the mode declarations

---

[1] If $\mu_i$ is the $i^{th}$ moment about the mean, then we define skewness and kurtosis as follows:

skewness = $\mu_3^2/\mu_2^3$

kurtosis = $\mu_4/\mu_2^2$

See [Kendall 63] for more details regarding the meaning of these measures.

for the input variables *vt* and *tl*, and to change the mode of the output to include *real\random*. The modified code is then the following:

```
Distto60MPH <-
  EXPR(vt:ONEOF(real\random, REAL),
    tl:ONEOF(real\random, REAL),
    d:REAL;
    ONEOF(real\random, REAL))
    BEGIN
      DECL kl:REAL BYVAL 6.076E3 / 3.6E3;
      DECL vf:REAL BYVAL 6.0E1 * 5.28E3 / 3.6E3;
      ( (vt * kl) ^ 2 - vf ^ 2) / (2.0 * d) + tl;
    END;
```

This code can then be directly executed to obtain the distribution of the 60-MPH point along the runway.

```
->Distto60MPH( uniform(98.0, 128.0), uniform(1000.0, 2000.0), 5.0)$
mean = 4.3843339E3 var = 3.9454025E5 skewness = 6.4335654E-2 kurtosis = 2.2038808
```

In fact, the touchdown velocity and location are not likely to change despite introduction of high-speed turnoffs, while the deceleration rate may be varied easily by the pilot over the range from 3 ft/sec/sec to 14 ft/sec/sec. This effect may be observed today as pilots "shoot" for specific turnoffs and decelerate just hard enough to make that turnoff.

We can then write a driver program which will step the deceleration rate over the range from 3 to 14 ft/sec/sec and print a table of results, which will operate correctly for either random variables or real variables for *vt* and *tl*.

```
Driver <-
   EXPR(vt:ONEOF(real\random, REAL), tl:ONEOF(real\random, REAL) )
     BEGIN
       PRINT('
d    60 MPH Location
');
       FOR i FROM 3 TO 14
     REPEAT
       PRINT(i);
       PRINT('        ');
       PRINT(Distto60MPH(vt, tl, i));
       PRINT('
');
    END;
      END;
```

This may then be executed to obtain the following tables:

```
->Driver( 113.0, 1500.0 );
d         60 MPH Location
3         6.2716158E3
```

```
4    5.0787119E3
5    4.3629695E3
6    3.8858079E3
7    3.5449782E3
8    3.2893559E3
9    3.0905386E3
10   2.9314847E3
11   2.8013498E3
12   2.692904E3
13   2.6011421E3
14   2.5224891E3

->Driver ( uniform(98, 128), uniform(1000, 2000) );

d    60 MPH Location
3    mean = 6.3072231E3 var = 9.477965E5 skewness = 7.9995624E-2 kurtosis = 1.99806
4    mean = 5.1054174E3 var = 5.69594E5 skewness = 7.2433547E-2 kurtosis = 2.1047709
5    mean = 4.3843339E3 var = 3.9454025E5 skewness = 6.4335654E-2 kurtosis = 2.2038808
6    mean = 3.9036116E3 var = 2.9944912E5 skewness = 5.6316626E-2 kurtosis = 2.2855201
7    mean = 3.5602385E3 var = 2.421125E5 skewness = 4.8762627E-2 kurtosis = 2.3447075
8    mean = 3.3027087E3 var = 2.0489863E5 skewness = 4.194312E-2 kurtosis = 2.3819054
9    mean = 3.1024077E3 var = 1.7938487E5 skewness = 3.597244E-2 kurtosis = 2.3989007
10   mean = 2.9421669E3 var = 1.6113513E5 skewness = 3.0791744E-2 kurtosis = 2.4008041
11   mean = 2.8110608E3 var = 1.4763231E5 skewness = 2.6383154E-2 kurtosis = 2.3915575
12   mean = 2.7018058E3 var = 1.3736231E5 skewness = 2.2648378E-2 kurtosis = 2.3736388
13   mean = 2.6093592E3 var = 1.2936988E5 skewness = 1.9486725E-2 kurtosis = 2.3510717
14   mean = 2.5301193E3 var = 1.2302806E5 skewness = 1.6836281E-2 kurtosis = 2.3252265
```

## Implementation of STAT

As indicated in Sec. III, the major portions of a statistical compiler system which differ from a conventional compiler are the representation choices and the simplification rules. In addition, the basic pieces of lexical and syntactic analysis must be provided. Thus, there were three major design choices to be made in the development of the STAT system: (1) the representation, (2) the approach to simplification, and (3) the mechanisms for lexical and syntactic analysis.

For the first two questions, we have extensively discussed the options in the previous sections. In the STAT system, only one representation form is employed: moments. This then implies that the operations conveniently available for pairs of *real\random* variables are multiplication and addition. Further, it is easy to provide addition and multiplication by a REAL, and exponentiation of a *real\random* to a positive integral power. These are thus the operations that have been provided to the STAT user.

For simplification, I have chosen to employ primarily the separating set rule. Phrased in a more intuitive and specific fashion than in Sec. IV, the rule states that if a set of variables can be found which separates the root of the computation tree from the terminal nodes, then the only information required to compute the distribution of the variable associated with the root node is the joint distribution of the variables in the separating set. In order to avoid keeping large multidimensional probability distributions, we limit our selection of separating sets to sets with independent variables. Thus, the only information we must maintain as we compute through a tree is the moments of the individual variables in the successive separating sets.

A key part of the compiler thus determines a separating set of independent variables for a given root node. This is employed recursively starting with the desired output node. That is, first a separating set of independent variables is determined for the output node. To obtain the moments for each non-terminal node in this set, the same routine is invoked recursively to determine a separating set for that node. This continues until the separating set determined consists entirely of terminal nodes whose moments can be directly evaluated.

One way to view this process is to say that STAT compiles backward through the program and then executes forward. In order to determine which variables form the necessary separating sets, STAT must start from the desired output variable. It can then move back to the variables in the separating set chosen for the output variables, and do the same operations as if each variable in this set were the output variable. When this process has pushed the attention of the STAT compiler up the tree to the terminal nodes, it may now begin evaluation of moments for the variables in the separating sets, moving down in the tree until the moments for the output variable have been evaluated.

The ECL system environment is used to aid in the construction of the computation tree for the simplification and evaluation sections of the STAT compiler. Such tasks as lexical and syntactic analysis of the source program are performed by the ECL interpreter. In addition, all operations on modes such as REAL, INT, BOOL, etc., available in ECL are directly accessible to the STAT user. These include conditionals, looping, subroutine invocation, and I/O functions. So long as *real\random* variables are not employed, the STAT routines are not invoked. Also, if a STAT user tries to perform an operation on a *real\random* which is not supported in STAT, then he will receive a MODE ERROR from ECL.

When the ECL program does reach supported operations on *real\random* variables, the STAT mechanisms are invoked. Unless a print operation is requested, the STAT routines merely use these calls to construct the computation tree. Each *real\random* variable is represented by a data structure which is a node in the computation tree. Calls to a STAT operation are used to link the nodes into the tree structure with pointers to the left and right fathers of the results, and to record the function associated with the node in the data structure. Thus, the STAT routines partially decompile the ECL program to obtain the structure (computation tree) of operations on *real\random* variables after all the operations on REALs, INTs, and BOOLs have been performed. All loops have been unwound, and all subroutine calls involving *real\random* variables have been expanded into one large computation tree.

When the PRINT routine is invoked for a *real\random* variable, that variable is designated as the output variable, and the STAT simplification and evaluation code is invoked. As described earlier, a separating set of independent variables is determined for the output variable.

The output variable is then expressed symbolically as a multinomial function of the variables in the separating set. This is done by backward substitution beginning with the output variable. The output variable can be expressed as some function, say addition, of two other random variables. These in turn can be expressed in terms of their fathers. Finally, the expansion will reach members of the separating set and the substitution will terminate.

If the moments of the variables in the separating set are available, then the moments of the output variable may be obtained directly by an evaluation process on the multinomial representation. Note this is not the normal evaluation of a multinomial; rather, as described in Sec. V, we must evaluate a term $X^iY^j$ as $E(X^i)E(Y^j)$. To obtain the higher moments of the output variable, we must symbolically raise the multinomial to a power to obtain the multinomial expansion for the higher moments.

After evaluation of a particular output variable, all the intermediate results are retained in the computation tree data structure. The motivation for this derives from the fact that the ECL user is operating interactively and may ask for further operations invoking the variables he has used. It would clearly have been possible to choose not to retain these intermediate results, and to regenerate them as needed.

Note that the moments of the input variables are not needed until after all the symbolic polynomial manipulations are performed. Moreover, once this analysis of the program structure has been performed, different input distributions could be specified to generate different output distributions without changing the intermediate analysis. STAT has not been organized to permit respecifying the input distributions, but again this could clearly have been done. In this sense, STAT is a compile and execute system rather than a compiler.

TABLE VI-1

STATIC BREAKDOWN OF STAT CODE

| | Lines of Code | Percent |
|---|---|---|
| ECL Interfoce and Computation Tree Construction | 195 | 27.5 |
| Simplification Strategy ond Evaluation Control | 110 | 15.5 |
| Polynomiol Monipulation Pockage | 180 | 25.4 |
| Set Manipulation Packoge | 40 | 5.6 |
| Constructor Definitions | 185 | 26.1 |
| TOTAL | 710 | |

The static breakdown of the code in the STAT system, shown in Table VI-1, provides some indication of where the programming effort was spent. The polynomial manipulating package represents the weakest portion of STAT, since the implementation is done using a list-structured representation for the polynomial and recursive calls to perform the required operations. This led to ease in coding, but requires long execution times.

This is reflected in the dynamie code breakdown shown in Table VI-2. These data were obtained on the Harvard PDP-10 (a KA-10 processor, earliest of the PDP-10 series), using the ECL interpreter to exeeute STAT when performing the statement

->Distto60MPH(uniform(98.0, 128.0), uniform(1000.0, 2000.0), 5.0)$

TABLE VI-2

DYNAMIC BREAKDOWN OF STAT CODE

| | CPU Time* (sec) | Percent |
|---|---|---|
| 1. Computation Tree Formation | 1.71 | 5 |
| 2. Separoting Set Selection | 0.81 | 2 |
| 3. Polynomiol Formation | 2.20 | 6 |
| 4. Raising Polynamials to Powers | 25.83 | 72 |
| 5. Evoluating Moments | | |
| A. Terminol Nodes | 1.58 | 4 |
| B. Intermediote Nodes | 3.43 | 10 |
| 6. Miscellaneous | 0.49 | 1 |
| TOTAL | 36.05 | 100 |

*Preliminary dota-individual values have voriance of 20 percent about their meon.

There are a number of interesting points to note in Table VI-2. Only about 15 pereent of the work (items 5A and 5B) depends on the moment values. Thus, if different input distributions are specified, it would be possible to recaleulate the results with only about 15 percent of the effort required for the first. The STAT system, as noted earlier, does not currently provide this option.

The other point of particular interest is that the time to perform the operation of raising the polynomials to a power and evaluating the moments could be easily estimated based on the number of terms in the polynomials formed by substitution. Thus, after about 15 pereent of the effort had been expended (steps 1, 2, and 3), the time for the remaining steps eould be predicted, and even presented as a table depending upon the number of moments desired in the output.

A listing of the STAT system and a more detailed description of the eode may be found in the Appendix.

# VII. PROBLEMS IN PROVIDING A PRACTICAL STATISTICAL COMPILER

In the last section, we described a statistical compiler system with a limited capability. In this section, we emphasize the limitations of the STAT system and indicate some approaches to the construction of a "complete" statistical compiler system.

The major limitation of STAT is the small variety of operations it is prepared to perform on random variables. As we discussed extensively in Sec. V, there is a complex interaction between the choice of representation form and the operations which are provided to the STAT user. By restricting our prototype to only a few operations, we were able to choose a representation which is convenient, accurate, and efficient. Extending to other operations however will involve compromises in some or all of these areas.

For example, assume that we wished to permit a random variable to appear in an arithmetic comparison with a real. This implies evaluating the cdf for the random variable at the value of the real. As we indicated in Sec. V, this conversion is expensive and error-prone unless the real value is in the tails of the distribution.

Of course, if the operations we wish to perform on the random variables are exclusively comparison, and maximum/minimum operations, then we should choose the cdf as the representation form. In this way, we could construct an alternative STAT system optimized for different operations and convenient for those.

We can pursue this further and construct a number of different statistical compilers each using a representation optimized for a particular set of operations. This set of statistical compilers would each handle a subset of all the problems one would like a statistical compiler to handle.

One can proceed a step further in the direction explored by Low [Low 74]. A master statistical compiler could be constructed which would contain all the particular representation forms we have the knowledge and patience to implement. The performance characteristics of the particular representations would be parametrically described, and the master compiler would then choose the representation form to minimize the resources required at execution time.

Following Low, there are a number of possible ways to proceed in the construction of such a master compiler. We might insist that conversions between representations are expensive and will not be permitted. In this case, we "partition" the random variables according to the operations performed on them. For each random variable, if the set of operations to be performed is small, then there may well be an optimum representation employed. In Low's study for representing sets, this technique of choosing one representation to use for the life of the variable was satisfactory and such a compiler was constructed.

In the statistical compiler area, because the costs of using the wrong representation are severe, and because no one representation is satisfactory, we may be forced to converting from one representation form to another. As Low properly indicates, such a compiler would at this time be a research effort in itself and constitutes an interesting problem.

There are several further possibilities beyond those suggested that might be used in a statistical compiler. One option of interest in a statistical compiler is the possibility of postponing computational effort as long as possible because the intermediate results can be so large. Indeed, it may in some cases be more economical to recalculate the results as needed than to save the result in storage. Thus, one set of representation options would be to produce subroutines which will calculate the values of specific entries in the representation form rather than to produce the representation form in toto.

Because any specific set of representations, such as those discussed in Sec. V, will not be convenient for some base functions, it is necessary to provide some alternate mechanism to handle these operations. The obvious alternative, first suggested in this context by Bērziņš [Bērziņš 75], is to employ Monte Carlo simulation for small sections of the program, and convert the various representation forms to and from a sampling representation. A master statistical compiler with this option can guarantee to be able to handle any "FORTRAN"-type program presented to it using random variables, although there will surely be some for which the errors or execution time will be unacceptable.

The master compiler can and should offer its user an estimate of the errors and execution time of his program. In the case of STAT, we could easily have presented the user with a fairly good prediction (±20 percent) of the running time of his program after only 15 percent of the total effort had been expended; we could have gone further and presented the user with a table of running times as a function of the number of moments desired in the output. Since the execution time may be large for cases of practical interest, this would permit the user to carefully consider the value of the information and compare it with the cost of obtaining this information. The analysis to calculate the running time can be performed rapidly enough, even for large programs, to be performed interactively, while the large execution time could be deferred for a batch environment, with the user knowing what resources he will be expending.

A practical statistical compiler must offer its user a variety of output options for their random variables. It must be prepared to provide moments about the origin, moments about the mean, cdf, or pdf. This goes directly back to the Sec. V discussion on representation conversions and the inherent accuracy limitations of some of these conversions. The user must be warned of the possible inaccuracies of his result in quantitative terms. It is not acceptable in an operational system to be used by many users in diverse situations to indicate "WARNING: CONVERSION FROM A MOMENT REPRESENTATION TO A CDF MAY PRODUCE SIGNIFICANT ERRORS IN THE OUTPUT." The system, if it is to be used, must indicate the location of the potential error and its magnitude. It must indicate what options the user has for avoiding the error (i.e., request your output as moments, not cdf). The system can and should track errors generated in intermediate operations and indicate the effects of their propagation. Again, this prescription becomes a significant research problem.

A less attractive alternate is to construct subsets of such a master compiler which can detect errors of significant proportions, and then insist on using higher accuracy in this subsection of the program (i.e., more samples in a Monte Carlo approach, higher moments in a moment representation, more points in a cdf, etc.). This approach can provide some assurance of adequate error control although not as reliably as tracking errors. It may require large or excessive amounts of computing to reduce errors which would not propagate further.

Another problem that the designer of a master statistical compiler must consider is the handling of user-written subroutines. In STAT, this is handled by, in effect, expanding each subroutine invocation into the in-line code which it represents, and then processing the whole program at once. This is unacceptably cumbersome for a compiler intended to handle programs of significant size. A practical statistical compiler must be capable of "separately compiling" subroutines. It should produce, in the run-time environment, a section of code which represents the subroutine and which is invoked on each call of the subroutine. This representation accepts distributions for those arguments whose data type is *random*, and produces a result which may be of type *random*.

The problem here is relatively simple if the input variables are statistically independent. In that case, the problem is precisely the same as compiling the main program, and is thus just the problem that the compiler is designed to cope with. However, the possibility that the input variables may not be independent complicates the construction of the subroutine code. Although the techniques indicated in Secs. IV and V are relevant to this problem, some of the simplifications were obtained by the assumptions of independence of the input variables. Relaxing this assumption makes simplifying the program structure more difficult. If all input variables are mutually dependent, then no effective simplification of the program can be performed. As more mutually independent sets of input variables are identified, more simplification can be performed.

Another problem in the construction of a statistical compiler is the question of an optimum computer architecture for the execution of such a compiler. We distinguish this into two phases: compile time and run time. In compile time activities, the activities of a statistical compiler are not significantly different from a regular compiler. There are some differences related to spending more time searching computation trees and performing algorithms on these trees. In addition, a relatively small percentage of the total time is spent in the compile time activities of the statistical compiler. The possible useful hardware modifications are in the areas of improved character handling and stack operations, as well as the possibility of special-purpose hardware for algorithms on tree structures. The special tree hardware might be capable of the following operations:

(1) Data Entry

x is father of y
x is son of y
x is brother of y
etc.

where x is being entered into the tree, and y is already a member.

(2) Data Retrieval

the set of sons of x
the set of ancestors of x
the set of common ancestors of x and y

where x and y are elements of the tree.

(3) Predicates

is x an ancestor of y?
are x and y independent?
does the set {z} separate x from y?

where {z} is a set of elements of the tree, and x and y are elements of the tree.

This tree manipulation hardware would be generally useful beyond just the statistical compiler. I see two possible implementation approaches to such hardware.

(1) Via a microprocessor operating on a private memory with algorithms optimized for the tree task using random access memory and pointers in the structure and,

(2) Via an associative memory using content addressing to rapidly retrieve items with the desired relationships.

Option (1) has the advantage of being fairly easy to construct today with commercially available hardware, while option (2) should be faster but is significantly more expensive.

The architecture for support of the run time system is the more important issue, and is crucially dependent on the representation forms used. Each form has some hardware structure that we could describe to optimize its execution; some of these are commercially available. For example, operations on pdf's might be performed using hardware convolvers and FFT boxes. These devices are now available commercially and will become cheaper over time. For example, convolvers on a single chip of CCD or MNOS technology are just now appearing in the laboratory [Tiemann 74].

If moment representations are to be employed, then hardware for symbolic manipulations of polynomials might be appropriate. I have no specific insight into the construction of such hardware, but if available it could be utilized for a statistical compiler as well as in a symbolic manipulation system [CACM 71].

In general, the analysis of programs for a statistical compilation reveals sections which can be pursued in parallel. This property of the analysis can be used to exploit hardware parallelism for those programs which are intrinsically parallel. These parallel sections follow closely disjoint sections of the underlying computation tree which permit easy determination of the necessary control structure. Thus, the picture I have of a processor designed for a statistical compiler run time system would have the following major features:

(1) A set of independent processors, not necessarily identical. Some of these would be optimized for convolution operations, some for symbolic polynomial manipulation, some for Monte Carlo simulations, etc. This is probably best handled using microprocessors with writable microcode so they can be rapidly optimized to handle the particular representation form for the assigned section of the tree.

(2) A master controller following a tree description of the control structure necessary for the execution of the program. The control structure would be constructed at compile time, and the controller would then see to initiating and responding to the terminations of processors.

This description leaves open the question of memory organization to support such a collection of processors, although this is likely to be critical to the success of a hardware implementation. If the processors are physically clustered, then a common multiport, highly interleaved memory preserves the flexibility to alter memory allocation dynamically. Constructing such a complicated memory bus will however surely occupy a large fraction of the implementation effort. The alternate approach of providing each processor with private memory is less attractive in this case because of the large and variable demands upon the memory resources in any particular execution.

This hardware organization would be interesting and useful for a broad variety of tasks, and thus could be made economically viable. The closest approach to this structure to my knowledge is the C.MMP effort at Carnegie [Wulf 72] which uses multiple PDP-11's rather than microprocessors. The design, implementation, and software for such a structure remains a fascinating and challenging problem.

# VIII. REMAINING RESEARCH PROBLEMS

In this document, we have presented a new philosophy and approach to computing with random variables. The work presented is, however, only a beginning in this area and much fruitful work remains to be performed. In this section, we indicate some of the major research problems that remain to be solved.

## A. OTHER APPROACHES

In Sec. II, we presented a non-computability result indicating some fundamental limitations on the types of computations which can be performed. As we indicated immediately following the theorem, there are a number of possible problems to be solved within these limitations. In the remainder of the report we have only explored one subproblem, but the solution of some of the others would be a useful addition to the state of the art.

TABLE VIII-1

| | Algorithm Type | | | | | |
|---|---|---|---|---|---|---|
| | Exact | | | Approximate | | |
| Distribution Class / Function Class | Continuous | Infinite Atomic | Finite Atomic | Continuous | Infinite Atomic | Finite Atomic |
| Hard | X | X | ? | MC | MC | MC |
| Easy | SC | SC | SC | MC | MC | MC |

SC = Statistical compiler techniques
MC = Monte Carlo techniques
X = Non-computable
? = No known practical techniques

As indicated in Table VIII-1, we can divide the problems into several classes. One categorization describes the class of functions the statistical compiler can handle. For this discussion, I divide the classes of functions loosely into "easy" classes and "hard" classes, where "easy" classes do not include Kleene's predicate, while "hard" classes do. The class of computation trees is considered an "easy" class in this sense. Another division of statistical compilers is based on whether the input distributions are atomic with a finite number of atoms, atomic with an infinite number of atoms, or continuous. The simplification rules described in Sec. IV apply to all these types of distributions, while the representation techniques in Sec. V emphasize the last two. Finally, a third division is based on whether the techniques used for the statistical compiler are capable of providing exact results assuming the computing device performs exact arithmetic. The approach in this work has been exact solutions, as contrasted, for example, with Monte Carlo techniques which are inherently approximate.

The table indicates which approaches have been pursued so far. As can be seen, the table has a major gap in practical techniques for the "hard" class of functions and finite atomic distributions. The only approaches currently available are exhaustively evaluating the function for all atoms of the input distribution (not practical) or sampling the input distribution (i.e., Monte Carlo technique). Some intermediate approach which analyzes the structure of the computation to reduce the amount of work seems fruitful. However, the non-computability results imply that any such techniques must critically depend on the finite nature of the distributions. The key question then is whether such algorithms will be combinatorially explosive for most practical problems or useful for a large class of functions.

The other problem indicated in the table is the extensive dependence on Monte Carlo and statistical compiler techniques for many different problems. I believe that more efficient approaches to the "easy" finite cases can be designed which will permit more accurate results than Monte Carlo techniques but which are less accurate than the statistical compiler techniques described here.

## B. REPRESENTATION TECHNIQUES

The analysis of representation techniques in Sec. V only begins to approach an exceedingly complex topic which is at the heart of statistical compilation techniques. The discussion in Sec. V suggests a large variety of techniques, but only a few of these have had significant utilization. More extensive experience with these techniques is clearly necessary to evaluate their suitability for use.

TABLE VIII-2

| Base Operations | Representotion Techniques |
|---|---|
| +, − | Chorocteristic Functions |
|  | Sampled pdf |
|  | Moments |
| * | Moments |
|  | Mellin Tronsform |
|  | Sampled pdf |
| MAX, MIN, CHOICE | cdf |
| SUM | Generoting Functions |

Table VIII-2 indicates a number of base functions and some representations which are particularly convenient for those base functions. The table shows that for the major operations a number of representation choices are available. However, the list of base functions here is relatively short, excluding such important operations as division and exponentiation. Further, the representation choices tend to be specific to certain base operations. Thus, there are two major areas of further effort: (1) new representations appropriate to other base operations, and (2) new algorithms for f-convolution for specific representations (i.e., a convenient algorithm for ÷ − convolution on sampled pdfs).

Further work in the extension of f-convolution algorithms to joint probability distributions could significantly improve computational speed and accuracy for complicated programs.

## C. APPLICATION AREAS

We began this report with two examples of applications for statistical compiler techniques. In Sec. VI, we showed the use of the STAT system to solve one of these problems. These examples indicate some possible uses of a statistical compiler, but represent a very limited subset of the actual application areas.

In general, application areas for a statistical compiler of the type envisioned in Sec. VII have a number of distinguishing properties, or conversely, if a problem has these distinguishing properties, it is a candidate for use of statistical compiler techniques. The major characteristics of these problems are (a) there is a large population of interest (large enough for statistical measures to be accurate enough for practical purposes), (b) each element of the population has certain properties which are expressed numerically, (c) the distribution of the values of the properties is known for the population, and (d) a deterministic description of the behavior (also specified numerically) of an element of the population in terms of its properties is available. Then one may use statistical compiler techniques to calculate the distribution of the values of the behavior of the population. Moreover, the problem may be presented to the compiler in two parts: the behavioral description, and the property distributions.

However, these very general guidelines do not indicate the real limitations of the techniques. As we have indicated elsewhere in this report, the two major practical problems are: (1) the complexity of the behavior function, and (2) the joint dependencies of the input distributions.

How then is a potential user to decide whether to employ these techniques. If a statistical compiler as envisioned in Sec. VII is available, the user may ask the compiler for estimates of running time and accuracy. But this is partially begging the questions, for the real issue is: For how many useful problems would a statistical compiler produce accurate results with a plausible expenditure of resources?, or expressed more succinctly, Is it worth the effort to construct an extensive statistical compiler?

Unfortunately, at this point, no quantitative data to support an answer to these questions exist. It is clear from the work presented here, that there is a class of problems for which these techniques are applicable and efficient. There is also clearly more research work to be performed before the construction of a "practical" statistical compiler can be sized accurately.

Thus, the major unsolved question remaining in this area is its ultimate practicality. Some specific subquestions can be phrased to help the resolution of this question: How well can we characterize the execution time of some of the f-convolution operations? Are there specific application areas which would be satisfied with just the base operations for which convenient representations are known? How complicated a process is the rule-guided optimization? How much storage must be dedicated to store joint distributions for practical problems?

The answers to these questions will significantly aid in determining the future of this line of inquiry. My belief, at this point, is that these questions will take time to answer, but that the ultimate utility of statistical compiler techniques will be demonstrated.

# APPENDIX
# DESCRIPTION OF STAT CODE

The STAT system is constructed entirely in ECL, using the ECL mechanisms for mode extension [Wegbreit 74]. These mechanisms have been general enough to allow for all the extensions described in Sec. VI without any changes to the ECL system, although certain capabilities to properly control the construction of *real\random* objects have only appeared in the system recently. The files which the user "LOAD"s to invoke STAT consist of several primary components:

(1) A set of functions required by the ECL system to specify the behavior of the new mode *real\random*. These include functions for generation, conversion, assignment, selection, and printing of *real\random* objects.

(2) Extensions to the operators "+", "-", "*", "/", and "^" to handle *real\random* variables. In the standard cases, the already existing system routine is invoked, while *real\random* objects trigger entry to code written for the STAT system.

(3) Definitions of the constructors *point, distribution, uniform, gaussian,* and *poisson,*

(4) Definitions for a new mode called *poly*, used to represent polynomials of several variables, which is used in the evaluation of *real\random* expressions as described below, and

(5) Routines which control the amount of evaluation effort expended at each point during the execution of a program involving *real\random* objects.

The data structure used to represent *real\random* objects has a number of fields:

| | |
|---|---|
| *name* | SYMBOL |

A guaranteed unique name which is used internally to refer to this random variable.

| | |
|---|---|
| *leftfather* | PTR ( REAL, INT, *real\random*) |
| *rightfather* | |

These fields point to the values that were combined to produce the random variable. They may point to another random variable, or a REAL or INT value. Only two such fields are necessary because only operators on one or two variables have been provided, although the extension to handling functions of several variables merely requires additional father fields.

| | |
|---|---|
| *fn* | ROUTINE |

The operator used to combine the fathers to obtain this random variable.

| | |
|---|---|
| *mom* | VECTOR( 16, REAL) |

Space to hold the moments about the origin of this random variable. Initially, none of these values are calculated; rather they are generated as needed.

| | |
|---|---|
| *curlth* | INT |

The index of the last entry in the ***mom*** table which has been filled.

| | |
|---|---|
| *momgen* | PROC( PTR( *real\random*) ) |

A routine which calculates additional moments as necessary. For random variables which are generated by constructors, specific ***momgen*** functions are provided, e.g., ***poissonmomgen, gaussianmomgen***, etc. For all intermediate results, a routine called ***intermomgen*** is used.

| | |
|---|---|
| *deslth* | INT |

The index of the last entry in the ***mom*** table which should be filled when the ***momgen*** routine is executed.

| | |
|---|---|
| *data* | REF |

Some of the ***momgen*** routines associated with specific constructors require space to store their internal variables. The ***data*** field refers to a data structure for this purpose. The exact details of this data structure vary for different constructors.

| | |
|---|---|
| *anc* | PTR( *rrSET* ) |

A pointer to the set of ***real\random*** objects which contains all the computational ancestors of this random variable. Not calculated until needed.

When a constructor is invoked, it invokes the generation routine for ***real\random*** variables. The generation routine provides a blank data structure, with only the ***name*** field initialized. The constructor assigns appropriate values to the ***data*** and ***momgen*** fields and returns.

Operators also invoke the generation routine to obtain a blank structure, and using a routine called ***rrapply*** assign values to the fields ***fn, momgen, leftfather***, and ***rightfather***. No other calculations are performed at this time.

These steps cause the system to construct the computation tree for the computation as the user is assigning values to variables and executing his routines, whether interactively or from predefined functions. Until the ***print*** function is invoked on a ***real\random***, no attempt is made to evaluate its moments.

The ***print*** routine is also quite simple. When invoked on a ***real\random***, it checks ***curlth*** to see if four moments are available. If they are not, the ***print*** routine sets ***deslth*** to four, and invokes the ***momgen*** routine to calculate the additional moments. When this is complete, four moments are available and may be printed.

The heart of the system is then the computational routines invoked via the moment generating functions, as well as the control which decides which moments need to be explicitly calculated. We describe first the algorithms used for some of the constructor ***momgen*** routines to indicate the algorithms for these, and then proceed to describe the more interesting routines invoked by ***intermomgen*** for non-terminal nodes of the computation tree.

## Uniform Moment Generator

The moments about the origin of a uniform distribution from a to b may be directly obtained by integration.

$$\mu'_n = \int_a^b \frac{x^n}{(b-a)}\,dx$$

$$= \frac{b^{(n+1)} - a^{(n+1)}}{(n+1)(b-a)}$$

If a specific $\mu'_n$ is desired, this formula may be evaluated directly, but if all values of $\mu'_n$ from n = 1 to k are desired, then the following recursive scheme is more efficient. Define

$$f(n, a, b) = \frac{b^{(n+1)} - a^{(n+1)}}{b - a}$$

Then we have the following recursive definition of f, as may be readily verified by induction.

$$f(0, a, b) = 1$$

$$f(n+1, a, b) = a * f(n, a, b) + b^{(n+1)}$$

$$\mu'_n = f(n, a, b)/(n+1)$$

Thus, if we maintain the values of f(n, a, b), and $b^n$, we may calculate each higher moment for the cost of two multiplications, one division, and two additions as well as the cost of updating the values for f and $b^n$. In addition, two locations for the storage of f and $b^n$ are required. These are provided in the *data* portion of the *real\random* data structure.

## Gaussian Moment Generator

The moments about the mean for the Gaussian are easier to obtain in closed form than the moments about the origin. We have then the following equation:

$$\mu_n = \int_{-\infty}^{\infty} (x-m)^n e^{-(x-m)^2/2\sigma^2}\,dx$$

Let y = x − m to obtain

$$\mu_n = \int_{-\infty}^{\infty} y^n e^{-y^2/2\sigma^2}\,dy$$

We may integrate this by parts to obtain

$$\int_{-\infty}^{\infty} y^n e^{-y^2/2\sigma^2}\,dy = \left.\frac{y^{n+1} e^{-y^2/2\sigma^2}}{n+1}\right|_{-\infty}^{\infty} + \int_{-\infty}^{\infty} \frac{y^{n+2}}{(n+1)\sigma^2} e^{-y^2/2\sigma^2}\,dy$$

Since $\lim_{y \to \pm\infty} y^{n+1} e^{-y^2/2\sigma^2} = 0$, we have the following recursive relation:

$$\int_{-\infty}^{\infty} y^{n+2} e^{-y^2/2\sigma^2}\,dy = (n+1)\sigma^2 \int_{-\infty}^{\infty} y^n e^{-y^2/2\sigma^2}\,dy$$

or expressed in a more usable form:

$$\mu_0 = 1, \qquad \mu_1 = 0$$

$$\mu_{n+2} = (n + 1) * \sigma^2 * \mu_n$$

This implies, as one would expect, that all odd moments about the mean are zero.

In order to obtain moments about the origin, we make use of the following relationship between moments about the mean $\mu_n$ and moments about the origin $\mu'_n$ [Kendall 63].

$$\mu_n = \sum_{j=0}^{n} \binom{n}{j} \mu'_j(-\mu'_1)^{n-j}$$

We may rewrite this to obtain

$$\mu'_n = \mu_n - \sum_{j=0}^{n-1} \binom{n}{j} \mu'_j(-\mu'_1)^{n-j}$$

expressing the nth moment about the origin in terms of the nth central moment and other moments about the origin. The polynomial in $(-\mu'_1)$ is best evaluated by Horner's rule, calculating the combinatorial coefficients in the same loop.

## Moment Calculation for Derived Results

As described in Sec. V, we can evaluate the moments of a multinomial function of independent random variables directly from the moments of those independent random variables. It is this technique which is used to implement the evaluation of moments for derived values.

In order to evaluate the moments of a ***real\random*** variable X which is not a direct result of a constructor, we then proceed as follows:

(1) Find a set (referred to as an *lia* set for reasons described later) of independent random variables which separates (see separating set definition in Sec. IV) X from its terminal ancestors in the computation tree (i.e., from all the constructor-generated random variables which are computational ancestors of X).

(2) Express X as a multinomial function of the random variables in the *lia* set.

(3) Evaluate (perhaps by recursive calls) the moments of the random variables in the *lia* set.

(4) Use these moments to perform the evaluation of the moments for X.

In order to reduce the computational effort as much as possible, both in finding a multinomial representation and in evaluating the multinomial, it is desirable that the size of the *lia* set be as small as possible. Further, the set must consist of independent ancestors of X. Because of this we refer to this set as the least independent ancestor (*lia*) set. There is not a unique least ancestor set for any arbitrary random variable X in any computation tree. Moreover, generating a set which is guaranteed to be minimal is not a simple computation. We choose therefore to calculate a set which is guaranteed to be independent and a separating set, which insures the accuracy of the computation, but we cannot guarantee that it is minimal. Thus there is a routine in the system called *lia* which when applied to a random variable calculates an approximation to a least independent ancestor set, and it is this set we refer to as the *lia* set.

Once the *lia* set is determined, the routine *subst* is invoked. This routine calculates the multinomial function representation for the random variable X in terms of members of the *lia* set. *Subst* first calculates the multinomial representation of *X.leftfather* and *X.rightfather* in terms of the *lia* set of X. It then uses the symbolic polynomial manipulation package which is part of STAT to combine these two according to the appropriate operation in X.*fn*. In fact, since the routines which implement " "+", "−", "*", "/", and "^", also accept arguments of mode *poly*, they are invoked directly to perform the polynomial manipulations.

When the substitution phase is complete, each term in the polynomial is examined to determine the highest degree for each *lia* set member in the multinomial. For example, if the derived multinomial for X is $X = 3 * Z^2 + 2 * Y^3 + 4 * Z^2 * Y^2$, then the highest degree for Z in this multinomial is 2, and the highest degree for Y is 3. If the number of moments desired for X is k (the value in *X.deslth* ), then 2 * k moments for Z are required, and 3 * k moments for Y are needed. If this number of moments for these variables are not available, then their moment generating functions are invoked, perhaps recursively, to obtain them. Once all the needed moments are available, the multinomial may be evaluated to obtain the first moment of X. As described in Sec. V, moments of order k may be obtained by symbolically raising the multinomial function to the power k, and then evaluating the multinomial obtained. The evaluation rules for multinomials in this context are not the normal rules of substitution since no specific values exist for the random variables in the *lia* set. Rather, when a term involving $Z^k$ is evaluated, the k$^{th}$ moment of Z is substituted for $Z^k$.

By far, the bulk of the computational effort expended occurs during the symbolic evaluation of the powers of the multinomial representation. The amount of effort required could be easily estimated on the basis of the number of terms in the multinomial and used to allow the user of the system to decide on the number of moments he requires when presented with the cost to compute each moment. No such provision is currently provided, however.

The remainder of the system provides more conventional capabilities such as the polynomial and set manipulation packages used to implement the moment generating routines. These will not be further discussed here.

A listing of the code for the STAT system follows.

```
poly <- poly :: SEQ(STRUCT(coeff:REAL, var:PTR("term")));

term <- term :: SEQ(STRUCT(var:PTR("rrSTRUCT"), exp:INT));

INFIX("!<", 150);

!< <- <;

< <-
  EXPR(x:ONEOF(REAL, INT, term), y:ONEOF(REAL, INT, term); BOOL)
    BEGIN
      CASE(COVERS)[MD(x), MD(y)]
      [ARITH, ARITH] => x !< y;
      [term, term] =>
        BEGIN
          DECL lx:INT BYVAL LENGTH(x);
          DECL ly:INT BYVAL LENGTH(y);
          FOR i FROM 1 TO LENGTH(y)
            REPEAT
              i > lx => TRUE;
              x[i].var # y[i].var =>
                VAL(x[i].var.name.TLB) !<
                  VAL(y[i].var.name.TLB);
              x[i].exp # y[i].exp => x[i].exp !< y[i].exp;
              FALSE;
            END;
        END;
      TRUE => BREAK('TYPE ERROR IN <');
      END;
    END;

rrSTRUCT <-
  rrSTRUCT ::
    STRUCT(name:SYMBOL,
         leftfather:PTR(REAL, INT, "rrSTRUCT"),
         rightfather:PTR(REAL, INT, "rrSTRUCT"),
         momgen:PROC(PTR("rrSTRUCT") SHARED),
         fn:ROUTINE,
         curlth:INT,
         deslth:INT,
         mom:VECTOR(16, REAL),
         anc:PTR("rrSET"),
         data:REF);
```

```
rafn <-
  EXPR(s:real\random, t:real\random; real\random)
    BEGIN
      DECL lowers:real\random.UR SHARED LOWER(s);
      DECL lowert:real\random.UR SHARED LOWER(t);
      lowers.leftfather <- lowert;
      lowers.fn <- iden;
      lowers.momgen <- intermomgen;
      s;
    END;

iden <- EXPR(x:poly; poly) x;

rcfn <-
  EXPR(s:real\random, t:MODE; ANY)
    BEGIN
      t = NONE => NOTHING;
      COVERS(t, PTR(rrSTRUCT)) OR COVERS(t, REF) => LOWER(s);
      BREAK('CONVERSION FROM real\random -- TYPE ');
    END;

rsfn <-
  EXPR(s:real\random, a:ONEOF(INT, SYMBOL); ANY)
    [] BREAK('SELECTION ON real\random ') [];

rpfn <-
  EXPR(s:real\random, p:PORT; real\random)
    BEGIN
      DECL mom:VECTOR(16, REAL) SHARED LOWER(s).mom;
      eval(s, 4);
      DECL var:REAL BYVAL mom[2] - mom[1] ^ 2;
      PRINT('mean =', p);
      PRINT(mom[1], p);
      PRINT(' var =', p);
      PRINT(var, p);
      var <- ABS var;
      var # 0.0 ->
      BEGIN
        DECL alpha3:REAL BYVAL
          (mom[3] - 3 * mom[1] * mom[2] + 2 * mom[1] ^ 3) /
            var ^ 1.5;
        DECL beta2:REAL BYVAL
          (mom[4] - 4 * mom[1] * mom[3] +
             6 * mom[1] ^ 2 * mom[2] - 3 * mom[1] ^ 4) /
            var ^ 2;
        PRINT(' skewness =', p);
        PRINT(alpha3, p);
        PRINT(' kurtosis =', p);
        PRINT(beta2, p);
      END;
      s;
    END;
```

```
uniquenamectr <- 1;

uniquename <-
  EXPR(; SYMBOL)
    BEGIN
      DECL s:SYMBOL BYVAL
      HASH(SCONCAT('name', BASIC\STR(uniquenamectr <+ 1)));
      s.TLB <- ALLOC(INT BYVAL uniquenamectr);
      s;
    END;

rgfn <-
  EXPR(m:MODE, s:SYMBOL, l:ANY; ONEOF(real\random))
    BEGIN
      m # real\random => BREAK('GENERATION ERROR - real\random');
      s = "BYVAL" => point(l);
      s = "LIKE" OR s = "SHARED" =>
      BEGIN
        CASE(COVERS)[MD(l)]
          [real\random] => l;
          [PTR(rrSTRUCT)] => LIFT(l, real\random);
          [REF] MD(VAL(l)) = rrSTRUCT =>
            BEGIN
              DECL z:PTR(rrSTRUCT);
              z <- l;
              LIFT(z, real\random);
            END;
          TRUE => BREAK('CONVERSION TO real\random ');
        END;
      END;
      DECL x:real\random.UR BYVAL ALLOC(rrSTRUCT);
      x.name <- uniquename();
      s = "SIZE" => LIFT(x, real\random);
      BREAK('GENERATION ERROR');
    END;

real\random <-
  <* "real\random",
     UCF(rcfn),
     UAF(rafn),
     USF(rsfn),
     UPF(rpfn),
     UGF(rgfn),
     SUPUGF(TRUE) *> :: PTR(rrSTRUCT);
```

```
rrapply <-
  EXPR(f:SYMBOL,
       x:ONEOF(real\random, INT, REAL),
       y:ONEOF(real\random, INT, REAL, NONE);
       real\random)
    BEGIN
      DECL z:real\random;
      DECL lowerz:real\random.UR SHARED LOWER(z);
      lowerz.fn <- VAL(f.TLB);
      lowerz.momgen <- intermomgen;
      lowerz.leftfather <-
      BEGIN
        MD(x) = real\random => LOWER(x);
        ALLOC(MD(x) BYVAL x);
      END;
      lowerz.rightfather <-
      BEGIN
        MD(y) = real\random => LOWER(y);
        ALLOC(MD(y) BYVAL y);
      END;
      z;
    END;

INFIX("!^", 225, TRUE);

FLUSH(^);

INFIX("^", 225, TRUE);

!^ <- EXPONENTIATION;

^ <-
  EXPR(x:ONEOF(INT, REAL, real\random, poly),
       y:ARITH;
       ONEOF(INT, REAL, real\random, poly))
    BEGIN
      CASE(COVERS)[MD(x), MD(y)]
      [ARITH, ARITH] => x !^ y;
      [poly, INT] =>
        BEGIN
          DECL p:PTR(poly) BYVAL ALLOC(poly SIZE 1);
          p[1].coeff <- 1.0;
          p[1].var <- ALLOC(term SIZE 0);
          FOR i FROM 1 TO y
            REPEAT p <- ALLOC(poly LIKE x * VAL(p)) END;
          VAL(p);
        END;
      [real\random, INT] => rrapply("^", x, y);
      TRUE -> BREAK('^ TYPE');
      END;
    END;
```

```
INFIX("!/", 200, TRUE);

!/ <- QUOTIENT;

/ <-
  EXPR(x:ONEOF(INT, REAL, real\random, poly),
       y:ARITH;
       ONEOF(INT, REAL, real\random, poly))
    BEGIN
      CASE(COVERS)[MD(x)]
      [real\random], [poly] => x * (1.0 / y);
      TRUE => x !/ y;
      END;
    END;

INFIX("!*", 200, TRUE);

!* <- PRODUCT;

* <-
  EXPR(x:ONEOF(INT, REAL, real\random, poly, term),
       y:ONEOF(INT, REAL, real\random, poly, term);
       ONEOF(INT, REAL, real\random, poly, term))
    BEGIN
      DECL mx:MODE BYVAL MD(x);
      DECL my:MODE BYVAL MD(y);
      COVERS(ARITH, mx) AND COVERS(ARITH, my) => x !* y;
      COVERS(poly, mx) AND COVERS(ARITH, my) =>
      BEGIN
        y = 0 OR y = 0.0 => CONST(poly SIZE 0);
        BEGIN
          DECL z:poly BYVAL x;
          FOR i FROM 1 TO LENGTH(z)
            REPEAT z[i].coeff <- z[i].coeff * y END;
          z;
        END;
      END;
      COVERS(ARITH, mx) AND COVERS(poly, my) => y * x;
      COVERS(real\random, mx) AND COVERS(poly, my) OR
      COVERS(poly, mx) AND COVERS(real\random, my) OR
      COVERS(term, mx) AND COVERS(real\random, my) OR
      COVERS(real\random, mx) AND COVERS(term, my) OR
      COVERS(term, mx) AND COVERS(ARITH, my) OR
      COVERS(ARITH, mx) AND COVERS(term, my) =>
      BREAK('TYPE ERROR *');
      COVERS(term, mx) AND COVERS(poly, my) =>
      BEGIN
        DECL z:poly BYVAL y;
        FOR i FROM 1 TO LENGTH(y)
          REPEAT
            z[i].var <- ALLOC(term SHARED VAL(y[i].var) * x);
          END;
        z;
      END;
  COVERS(poly, mx) AND COVERS(term, my) => y * x;
  COVERS(term, mx) AND COVERS(term, my) =>
  BEGIN
   DECL lx:INT BYVAL LENGTH(x);
   DECL ly:INT BYVAL LENGTH(y);
   DECL z:term SIZE lx + ly;
   DECL ix:INT BYVAL 1;
   DECL iy:INT BYVAL 1;
   DECL j:INT BYVAL 0;
   REPEAT
    DECL copy:ROUTINE BYVAL
     EXPR(x:term SHARED, ix:INT SHARED, lx:INT SHARED)
      REPEAT
       ix > lx => NIL;
       j <+ 1;
       z[j] <- x[ix];
       ix <+ 1;
      END,
    ix > lx => copy(y, iy, ly);
    iy > ly => copy(x, ix, lx);
    BEGIN
     j <+ 1;
     VAL(x[ix].var.name.TLB) = VAL(y[iy].var.name.TLB) =>
      BEGIN
       z[j] <- x[ix];
       z[j].exp <+ y[iy].exp;
       ix <+ 1;
       iy <+ 1;
      END;
     VAL(x[ix].var.name.TLB) > VAL(y[iy].var.name.TLB) =>
      [) z[j] <- x[ix]; ix <+ 1 (];
     z[j] <- y[iy];
     iy <+ 1;
    END;
   END;
   BEGIN
    DECL oz:PTR(term) BYVAL ALLOC(term SIZE j);
    FOR i FROM 1 TO j REPEAT oz[i] <- z[i] END;
    VAL(oz);
   END;
  END;
  COVERS(poly, mx) AND COVERS(poly, my) =>
  BEGIN
   DECL z:PTR(poly) BYVAL ALLOC(poly SIZE 0);
   LENGTH(y) < LENGTH(x) => y * x;
   FOR i FROM 1 TO LENGTH(x)
    REPEAT
     z <-
      ALLOC(poly LIKE
         VAL(z) + x[i].coeff * (VAL(x[i].var) * y));
    END;
   VAL(z);
  END;
  TRUE => rrapply("*", x, y);
END;
```

```
INFIX("!-", 175, TRUE);

PREFIX("!-");

!- <- DIFF;

- <-
  EXPR(x:ONEOF(INT, REAL, real\random, poly),
       y:ONEOF(INT, REAL, real\random, poly, NONE);
       ONEOF(REAL, INT, real\random, poly))
    BEGIN
      CASE(COVERS)[MD(x), MD(y)]
      [ARITH, ARITH] => x !- y;
      [ARITH, NONE] => !- x;
      [poly, NONE] =>
        BEGIN
          DECL z:poly BYVAL x;
          FOR j FROM 1 TO LENGTH(z)
            REPEAT z[j].coeff <- !- (z[j].coeff) END;
          z;
        END;
      [real\random, NONE] => rrapply("-", x);
      TRUE => x + - y;
      END;
    END;

INFIX("!+", 175, TRUE);

!+ <- SUM;

+ <-
  EXPR(x:ONEOF(REAL, INT, real\random, poly),
       y:ONEOF(REAL, INT, real\random, poly);
       ONEOF(REAL, INT, real\random, poly))
    BEGIN
      CASE(COVERS)[MD(x), MD(y)]
      [ARITH, ARITH] => x !+ y;
      [poly, ARITH] =>
        BEGIN
          LENGTH(VAL(x[1].var)) = 0 =>
            [) DECL z:poly BYVAL x; z[1].coeff <+ y; z (];
          DECL z:poly SIZE LENGTH(x) + 1;
          z[1].coeff <- y;
          z[1].var <- ALLOC(term SIZE 0);
          FOR i FROM 1 TO LENGTH(x)
            REPEAT z[i + 1] <- x[i] END;
          z;
        END;
      [ARITH, poly] => y + x;
      [poly, poly] =>
        BEGIN
          DECL lx:INT BYVAL LENGTH(x);
          DECL ly:INT BYVAL LENGTH(y);
          DECL z:poly SIZE lx + ly;
      DECL ix:INT BYVAL 1;
      DECL iy:INT BYVAL 1;
      DECL j:INT BYVAL 0;
      REPEAT
        DECL copy:ROUTINE BYVAL
          EXPR(x:poly SHARED, ix:INT SHARED, lx:INT SHARED)
            REPEAT
              ix > lx => NIL;
              j <+ 1;
              z[j] <- x[ix];
              ix <+ 1;
            END;
        ix > lx => copy(y, iy, ly);
        iy > ly => copy(x, ix, lx);
        BEGIN
          j <+ 1;
          VAL(x[ix].var) = VAL(y[iy].var) =>
            BEGIN
              z[j] <- x[ix];
              z[j].coeff <+ y[iy].coeff;
              ix <+ 1;
              iy <+ 1;
              z[j].coeff = 0.0 => j <- j - 1;
            END;
          VAL(x[ix].var) < VAL(y[iy].var) =>
            [) z[j] <- x[ix]; ix <+ 1 (];
          z[j] <- y[iy];
          iy <+ 1;
        END;
      END;
      BEGIN
        DECL oz:PTR(poly) BYVAL ALLOC(poly SIZE j);
        FOR i FROM 1 TO j REPEAT oz[i] <- z[i] END;
        VAL(oz);
      END;
    END;
  [poly, real\random], [real\random, poly] =>
    BREAK('type error +');
  TRUE => rrapply("+", x, y);
  END;
END;
```

```
eval <-
  EXPR(u:PTR(rrSTRUCT), k:INT)
    [) u.deslth <- k; u.deslth > u.curlth => u.momgen(u) (];

rrSET <- rrSET :: STRUCT(members:SEQ(BOOL), ptrs:PTR("rrMEM"));

rrMEM <- rrMEM :: STRUCT(elem:PTR(rrSTRUCT), next:PTR("rrMEM"));

INFIX("element", 150);

insert <-
  EXPR(x:PTR(rrSTRUCT), y:rrSET SHARED; rrSET)
    BEGIN
      x element y => y;
      VAL(x.name.TLB) > LENGTH(y.members) => BREAK('INSERTION ');
      y.members[VAL(x.name.TLB)] <- TRUE;
      y.ptrs <- ALLOC(rrMEM OF x, y.ptrs);
      y;
    END;

element <-
  EXPR(x:PTR(rrSTRUCT), y:rrSET; BOOL)
    BEGIN
      VAL(x.name.TLB) > LENGTH(y.members) => FALSE;
      y.members[VAL(x.name.TLB)];
    END;

intermomgen <-
  EXPR(u:PTR(rrSTRUCT) SHARED)
    BEGIN
      DECL y:rrSET LIKE lia(u);
      DECL sym:poly LIKE subst(u, y);
      calcdeslth(sym, LOWER(u).deslth);
      DECL x:PTR(rrMEM) BYVAL y.ptrs;
      REPEAT
      x = NIL => NIL;
      x.elem.deslth > x.elem.curlth -> x.elem.momgen(x.elem);
      x <- x.next;
      END;
      evalpoly(u, sym);
    END;
```

```
lia <-
  EXPR(x:PTR(rrSTRUCT); rrSET)
    BEGIN
      x.leftfather = NIL AND x.rightfather = NIL =>
      [] findterm(x); VAL(x.anc) [];
      MD(VAL(x.leftfather)) # rrSTRUCT =>
      BEGIN
        DECL y:rrSET SIZE VAL(x.rightfather.name.TLB);
        insert(x.rightfather, y);
      END;
      MD(VAL(x.rightfather)) # rrSTRUCT =>
      BEGIN
        DECL y:rrSET SIZE VAL(x.leftfather.name.TLB);
        insert(x.leftfather, y);
      END;
      findterm(x);
      disjoint(x.leftfather.anc, x.rightfather.anc) =>
      BEGIN
        DECL y:rrSET SIZE VAL(x.name.TLB);
        insert(x.leftfather, y);
        insert(x.rightfather, y);
      END;
      VAL(x.anc);
    END;

findterm <-
  EXPR(x:PTR(rrSTRUCT))
    BEGIN
      x.anc # NIL => NIL;
      x.leftfather = NIL AND x.rightfather = NIL =>
      BEGIN
        x.anc <- ALLOC(rrSET SIZE VAL(x.name.TLB));
        insert(x, VAL(x.anc));
      END;
      MD(VAL(x.rightfather)) # rrSTRUCT =>
      [] findterm(x.leftfather); x.anc <- x.leftfather.anc [];
      findterm(x.rightfather);
      MD(VAL(x.leftfather)) # rrSTRUCT =>
      x.anc <- x.rightfather.anc;
      findterm(x.leftfather);
      x.anc <- union(x.leftfather.anc, x.rightfather.anc);
    END;
```

```
union <-
  EXPR(s1:PTR(rrSET), s2:PTR(rrSET); PTR(rrSET))
    BEGIN
      LENGTH(s1.members) > LENGTH(s2.members) => union(s2, s1);
      DECL sz:PTR(rrSET) BYVAL ALLOC(rrSET BYVAL VAL(s2));
      DECL s1p:PTR(rrMEM) BYVAL s1.ptrs;
      REPEAT
      s1p = NIL => NIL;
      insert(s1p.elem, VAL(sz));
      s1p <- s1p.next;
      END;
      sz;
    END;

disjoint <-
  EXPR(s1:PTR(rrSET), s2:PTR(rrSET); BOOL)
    BEGIN
      LENGTH(s2.members) > LENGTH(s1.members) =>
      disjoint(s2, s1);
      FOR i FROM 1 TO LENGTH(s2.members)
      REPEAT
        s2.members[i] AND s1.members[i] => FALSE;
        TRUE;
      END;
    END;

subst <-
  EXPR(u:PTR(rrSTRUCT), lia:rrSET SHARED; poly)
    BEGIN
      u element lia => polymake(u);
      u.fn(BEGIN
           MD(VAL(u.leftfather)) = rrSTRUCT =>
             subst(u.leftfather, lia);
           VAL(u.leftfather);
         END,
         BEGIN
           MD(VAL(u.rightfather)) = rrSTRUCT =>
             subst(u.rightfather, lia);
           VAL(u.rightfather);
         END);
    END;
```

```
calcdeslth <-
  EXPR(p:poly, k:INT)
    FOR i FROM 1 TO LENGTH(p)
      REPEAT
      DECL t:PTR(term) BYVAL p[i].var;
      FOR j FROM 1 TO LENGTH(t)
        REPEAT
          k * t[j].exp > t[j].var.deslth ->
            t[j].var.deslth <- k * t[j].exp;
        END;
      END;

evalpoly <-
  EXPR(u:PTR(rrSTRUCT), p:poly)
    BEGIN
      DECL pow:PTR(poly) BYVAL
      ALLOC(poly LIKE p ^ (u.curlth + 1));
      DECL mom:VECTOR(16, REAL) SHARED u.mom;
      FOR m FROM u.curlth + 1 TO u.deslth
      REPEAT
        m # u.curlth + 1 ->
          pow <- ALLOC(poly LIKE p * VAL(pow));
        mom[m] <- evalterm(mom[m], VAL(pow));
      END;
      u.curlth <- u.deslth;
    END;

evalterm <-
  EXPR(m:REAL, pow:poly SHARED; REAL)
    BEGIN
      FOR i FROM 1 TO LENGTH(pow)
      REPEAT
        DECL t:term SHARED VAL(pow[i].var);
        m <+
          pow[i].coeff *
            BEGIN
              DECL prod:REAL BYVAL 1.0;
              FOR j FROM 1 TO LENGTH(t)
                REPEAT
                  prod <- prod * t[j].var.mom[t[j].exp];
                END;
              prod;
            END;
      END;
    END;
```

```
makedist <-
  EXPR(datamd:MODE, mean:REAL, momgenf:ROUTINE; real\random)
    BEGIN
      DECL z:real\random;
      DECL lowerz:real\random.UR SHARED LOWER(z);
      lowerz.data <- ALLOC(datamd);
      lowerz.momgen <- momgenf;
      lowerz.curlth <- 1;
      lowerz.mom[1] <- mean;
      z;
    END;

point <-
  EXPR(x:REAL; real\random) [) makedist(NONE, x, pointmomgen) (];

polymake <-
  EXPR(x:real\random; poly)
    BEGIN
      DECL p:PTR(poly) BYVAL ALLOC(poly SIZE 1);
      DECL t:PTR(term) BYVAL ALLOC(term SIZE 1);
      p[1].coeff <- 1.0;
      p[1].var <- t;
      t[1].exp <- 1;
      t[1].var <- x;
      VAL(p);
    END;

pointmomgen <-
  EXPR(x:PTR(rrSTRUCT) SHARED)
    BEGIN
      FOR i FROM x.curlth + 1 TO x.deslth
      REPEAT x.mom[i] <- x.mom[1] * x.mom[i - 1] END;
      x.curlth <- x.deslth;
    END;

uniform <-
  EXPR(a:REAL, b:REAL; real\random)
    BEGIN
      DECL x:real\random SHARED
      makedist(STRUCT(a:REAL, b:REAL, bpow:REAL, sum:REAL),
               (a + b) / 2,
               uniformmomgen);
      DECL xdata:REF BYVAL LOWER(x).data;
      xdata.a <- a;
      xdata.b <- b;
      xdata.bpow <- b;
      xdata.sum <- a + b;
      x;
    END;
```

```
uniformmomgen <-
  EXPR(x:PTR(rrSTRUCT) SHARED)
    BEGIN
      DECL xdata:REF BYVAL x.data;
      FOR i FROM x.curlth + 1 TO x.deslth
      REPEAT
        xdata.bpow <- xdata.b * xdata.bpow;
        xdata.sum <- xdata.a * xdata.sum + xdata.bpow;
        x.mom[i] <- xdata.sum / (i + 1);
      END;
      x.curlth <- x.deslth;
    END;

nextmom <-
  EXPR(n:INT, mom:VECTOR(16, REAL) SHARED, m:REAL; REAL)
    BEGIN
      DECL u:INT BYVAL 1;
      DECL var:REAL BYVAL - mom[1];
      DECL sum:REAL BYVAL var;
      FOR i FROM 1 TO n - 1
      REPEAT
        u <- u * (n + 1 - i) / i;
        sum <- var * (sum + u * mom[i]);
      END;
      m - sum;
    END;

gaussian <-
  EXPR(mean:REAL, sigma:REAL; real\random)
    BEGIN
      DECL x:real\random SHARED
      makedist(STRUCT(pow:REAL, var:REAL),
               mean,
               gaussianmomgen);
      DECL xdata:REF BYVAL LOWER(x).data;
      xdata.pow <- 1.0;
      xdata.var <- sigma * sigma;
      x;
    END;

gaussianmomgen <-
  EXPR(x:PTR(rrSTRUCT) SHARED)
    BEGIN
      DECL xdata:REF BYVAL x.data;
      FOR i FROM x.curlth + 1 TO x.deslth
      REPEAT
        x.mom[i] <-
          BEGIN
            nextmom(i,
                    x.mom,
                    BEGIN
                      i / 2 * 2 = i =>
                        xdata.pow <-
                          (i - 1) * xdata.pow * xdata.var;
                      0;
                    END);
            END;
        END;
        x.curlth <- x.deslth;
      END;

  poisson <-
    EXPR(lambda:REAL; real\random)
      BEGIN
        DECL x:real\random SHARED
        makedist(STRUCT(p1:VECTOR(17, INT), p2:VECTOR(17, INT)),
                 lambda,
                 poissonmomgen);
        DECL xdata:REF BYVAL LOWER(x).data;
        xdata.p2[1] <- 2;
        xdata.p2[2] <- 1;
        xdata.p1[1] <- 1;
        x;
      END;

  poissonmomgen <-
    EXPR(x:PTR(rrSTRUCT) SHARED)
      BEGIN
        DECL xdata:REF BYVAL x.data;
        DECL p1:VECTOR(17, INT) SHARED xdata.p1;
        DECL p2:VECTOR(17, INT) SHARED xdata.p2;
        FOR i FROM x.curlth + 1 TO x.deslth
        REPEAT
          x.mom[i] <-
            BEGIN
              nextmom(i,
                      x.mom,
                      BEGIN
                        DECL sum:REAL BYVAL 0.0;
                        FOR j FROM p2[1] BY - 1 TO 2
                          REPEAT
                            sum <- (sum + p2[j]) * x.mom[1];
                          END;
                        FOR j FROM p1[1] BY - 1 TO 2
                          REPEAT p1[j + 1] <- p1[j] * i END;
                        p1[2] <- 0;
                        FOR j FROM p1[1] + 1 BY - 1 TO 2
                          REPEAT
                            DECL t:INT BYVAL p1[j];
                            p1[j] <- p2[j];
                            p2[j] <- (j - 1) * p2[j] + t;
                          END;
                        BEGIN
                          DECL t:INT BYVAL p1[1];
                          p1[1] <- p2[1];
                          p2[1] <- 1 + t;
                        END;
                        sum;
                      END);
        END;
      END;
      x.curlth <- x.deslth;
    END;

distribution <-
  EXPR(tab:SEQ(VECTOR(2, REAL)); real\random)
    BEGIN
      DECL z:real\random;
      DECL lowerz:real\random.UR SHARED LOWER(z);
      lowerz.data <- ALLOC(MD(tab) BYVAL tab);
      lowerz.momgen <- dismomgen;
      lowerz.curlth <- 0;
      z;
    END;

dismomgen <-
  EXPR(x:PTR(rrSTRUCT) SHARED)
    BEGIN
      FOR i FROM x.curlth + 1 TO x.deslth
      REPEAT
        DECL sum:REAL BYVAL 0.0;
        FOR j FROM 1 TO LENGTH(x.data)
          REPEAT
            sum <- sum + x.data[j][1] * x.data[j][2] ^ i;
          END;
        x.mom[i] <- sum;
      END;
      x.curlth <- x.deslth;
    END;

buildtab <-
  EXPR(x:SEQ(REAL); SEQ(VECTOR(2, REAL)))
    BEGIN
      DECL z:SEQ(VECTOR(2, REAL)) SIZE LENGTH(x) / 2;
      FOR i FROM 1 TO LENGTH(x) / 2
      REPEAT z[i][1] <- x[2 * i - 1]; z[i][2] <- x[2 * i] END;
      z;
    END;
```

## ACKNOWLEDGEMENTS

There are a number of people who have provided assistance of various types during this work; it is a pleasure to take this opportunity to acknowledge their efforts. Dr. William Sutherland of Xerox Palo Alto Research Center and Professor William Bossert of Harvard have had faith in this work, even at times when I did not. That reservoir of faith helped carry me over some of the more traumatic periods of development. Professor Ugo Gagliardi of Harvard has patiently commented on a number of drafts, while Professor Thomas Cheatham of Harvard had the wisdom to suggest the importance of an implementation. I had an extremely illuminating conversation with Professor Dionysios Tsitchritzis during the early stages of this work which led to the noncomputability result in Sec. 11. My friends, Professors Paul Rovner of the University of Rochester and Richard Basener of Lehigh University have provided me with many helpful discussions and comments on the early drafts of this document. My supervisors at M.I.T. Lincoln Laboratory over the years have been understanding of the dual responsibilities of student and employee. To the late Mr. Jack Mitchell, Mr. James Forgie, Mr. Jack Raffel, Dr. Bernard Gold, Dr. Melvin Herlin, and Mr. Alan McLaughlin go my sincere appreciation. The Publications Group at Lincoln has done a professional job of preparing this document; they have significantly eased the effort of the final stages. Many other colleagues, fellow graduate students, and teachers have been kind enough to discuss various issues from time to time; a collective thank you to all.

## REFERENCES

[Agarwal 75] R.C. Agarwal and C.S. Burrus, "Number Theoretic Transforms to Implement Fast Digital Convolution," in [IEEE 75].

[Aho 72] A.V. Aho and J.D. Ullman, The Theory of Parsing, Translation, and Compiling, Vol. 1: Parsing (Prentice-Hall, New York, 1972).

[Allen 75] J. Allen, "Special Purpose Hardware for Signal Processing," in [IEEE 75].

[Apostol 57] T.M. Apostol, Mathematical Analysis, A Modern Approach to Advanced Calculus (Addison-Wesley, Reading, Mass., 1957).

[Bateman 54] Bateman Manuscript Project, Tables of Integral Transforms, Vol. 1 (McGraw-Hill, New York, 1954).

[Bergland 69] G.D. Bergland, "A Guided Tour of the Fast Fourier Transform," IEEE Spectrum 6 (July 1969). Reprinted in [Rabiner 72].

[Bērziņš 75] V. Bērziņš, "Algorithms for Analyzing Statistical Models of Information Systems," MS and EE thesis, M.I.T., Cambridge, Massachusetts, January 1975.

[Burr 42] I.W. Burr, "Cumulative Frequency Functions," Ann. Math. Statistics 13 (1942).

[CACM 71] Issue containing papers from the Second Symposium on Symbolic and Algebraic Manipulation, Communications of the A.C.M., Vol. 14, No. 8 (August 1971).

[Cobham 64] A. Cobham, "The Intrinsic Computational Difficulty of Functions," Proceedings 1964 International Conference for Logic, Methodology and Philosophy, edited by Y. Bar-Hillel (North-Holland, Amsterdam, 1964), pp. 24-30.

[Cooley 65] J.W. Cooley and J.W. Tukey, "An Algorithm for Machine Calculation of Complex Fourier Series," Mathematics of Computation 19, 297-301 (April 1965).

[Cooley 67] J.W. Cooley, P.A.W. Lewis, and P.D. Welch, "Historical Notes on the Fast Fourier Transform," IEEE Trans. Audio Electroacoust. AU-15, 76-79 (1967).

[Cooley 70] J.W. Cooley, P.A.W. Lewis, and P.D. Welch, "The Fast Fourier Transform Algorithm: Programming Considerations in the Calculation of Sine, Cosine and LaPlace Transforms," J. Sound Vib. 12 (July 1970). Reprinted in [Rabiner 72].

[Davis 58] M. Davis, Computability and Unsolvability (McGraw-Hill, New York, 1958).

[Davis 75] P.J. Davis and P. Rabinowitz, Methods of Numerical Integration (Academic Press, New York, 1975).

[Ditkin 65] V.A. Ditkin and A.P. Prudnikov, Integral Transforms and Operational Calculus, translated from Russian by D.E. Brown (Pergamon Press, Oxford, 1965).

[Feller 57] W. Feller, An Introduction to Probability Theory and its Applications, two volumes, Second Edition (John Wiley, New York, 1957).

[Franson 69] A. Franson, "Prediction of Statistical System Performance from Parameter Distributions," MSEE thesis, Naval Postgraduate School, Monterey, California, June 1969. Available from Clearinghouse for Federal Scientific and Technical Information as AD-703258.

[Gold 69] B. Gold and C.M. Rader, Digital Processing of Signals (McGraw-Hill, New York, 1969).

[Hammersley 64] J.M. Hammersley and D.C. Handscomb, Monte Carlo Methods (Methuen, London, 1964).

[Harvard 74] ECL Programmer's Manual, Report No. 23-74, Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts, December 1974.

[IEEE 75] Proceedings of the IEEE, special issue on Digital Signal Processing, April 1975.

[Karlin 53] S. Karlin and L. S. Shapley, "Geometry of Moment Spaces," Memoirs of Am. Math. Soc. 12 (1953).

[Kendall 63] M. G. Kendall and A. Stuart, The Advanced Theory of Statistics, Vol. 1: Distribution Theory, Second Edition (Hafner, New York, 1963).

[Low 74] J. Low, "Automatic Coding: Choice of Data Structures," PhD thesis, Report CS-452, Stanford University, Stanford, California, August 1974.

[Meyer 67] A. R. Meyer and D. M. Ritchie, "The Complexity of Loop Programs," Proceedings of 22nd National Computer Conference, Association for Computing Machinery (Thompson, Washington, 1967), pp. 465-469.

[Moore 66] R. E. Moore, Interval Analysis (Prentice-Hall, New York, 1966).

[Parzen 60] E. Parzen, Modern Probability Theory and Its Applications (John Wiley, New York, 1960).

[Pearson 68] K. Pearson, Tables of the Incomplete Beta Function, Second Edition (Cambridge University Press, Cambridge, England, 1968).

[Rabiner 72] L. R. Rabiner and C. M. Rader, Editors, Digital Signal Processing (IEEE Press, New York, 1972).

[Rice 64] J. R. Rice, The Approximation of Functions, Vol. 1: Linear Theory (Addison-Wesley, Reading, Mass., 1972).

[Ritchie 63] R. W. Ritchie, "Classes of Predictably Computable Functions," Trans Am. Math. Soc. 106, 139-173 (1963).

[Sain 73] M. K. Sain, E. W. Henry, and J. J. Uhran, "An Algebraic Method for Simulating Legal Systems," Simulation 21, 150-158 (November 1973).

[Schaefer 73] M. Schaefer, A Mathematical Theory of Global Program Optimization (Prentice-Hall, New York, 1973).

[Shohat 43] J. A. Shohat and J. D. Tamarkin, "The Problem of Moments," in Mathematical Surveys, No. 1 (American Mathematical Society, New York, 1943).

[Standish 75] T. A. Standish, "Extensibility in Programming Language Design," Proceedings of National Computer Conference 1975, Vol. 44 (AFIPS Press, Montvale, New Jersey, 1975), pp. 278-290.

[Stockham 66] T. Stockham, Jr., "High-Speed Convolution and Correlation," Proceedings of the Spring Joint Computer Conference 1966, Vol. 28 (Spartan Books, Washington, 1966) pp. 229-233.

[Stockham 69] T. Stockham, Jr., "High-Speed Convolution and Correlation with Applications to Digital Filtering," Chapter 7 in [Gold 69].

[Stroud 71] A. H. Stroud, Approximate Calculation of Multiple Integrals (Prentice-Hall, New York, 1971).

[Tiemann 74] J. J. Tiemann, N. E. Engeles, and R. D. Baertach, "A Programmable Transversal Filter using Charge Transfer Devices," Proceedings of the IEEE International Convention and Exposition, New York, 26-29 March 1974.

[Wegbreit 74] B. Wegbreit, "The Treatment of Data Types in EL1," Communications of the A. C. M., Vol. 17, No. 5 (May 1974).

[Wilkinson 63] J. H. Wilkinson, Rounding Errors in Algebraic Processes (Prentice-Hall, New York, 1963).

[Wulf 72] W. A. Wulf and C. G. Bell, "C.MMP-A Multi-mini-Processor," Proceedings of the AFIPS Fall Joint Computer Conference 1972, Anaheim, California, 5-7 December 1972, pp. 765-777.